☕ Teaching Broad Reasoning Skills for Multi-Step QA by Generating Hard 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 hard-to-cheat synthetic contexts for real questions in six multi-step 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 multi-step 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 F1 points across 4 multi-step QA datasets, with up to 21 point gain on more complex questions. The resulting models also demonstrate higher robustness, with a 5-8 F1 point improvement on two contrast sets. Furthermore, TeaBReaC pretraining substantially improves model performance and robustness even when starting with numerate LMs pretrained using recent methods (e.g., PReasM, POET). Our work thus shows how to effectively use decomposition-guided contexts to robustly teach multi-step reasoning.[1]

1 Introduction

Multi-step 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 needed for the question. However,
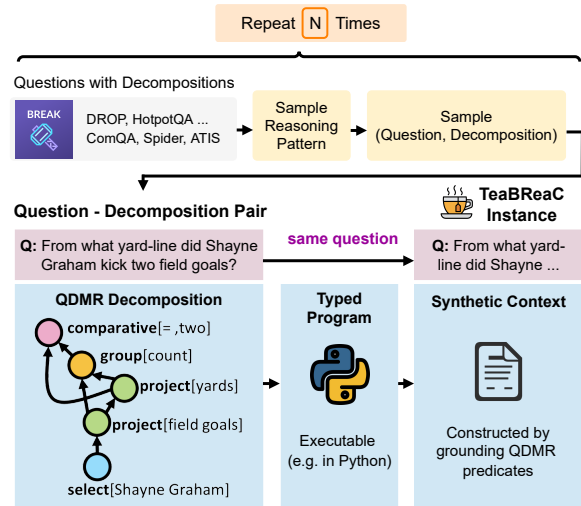

Figure 1: TEABREAC ☕ Dataset Construction: We leverage widely available question decomposition annotations (QDMRs) for real questions from a broad range of datasets to carefully construct synthetic contexts such that answering the resulting ☕ question requires proper multi-step reasoning. These questions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

even though questions in multi-step datasets often cover a broad range of interesting reasoning patterns, most questions follow only a few patterns, which is what models trained on these datasets 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.

How, then, can we teach models broad multi-step reasoning skills? One way is to have greater control over the distribution of reasoning patterns and the types of input contexts models see during training—contexts that don't allow 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 reason-

ing patterns. The challenge, then, is to teach these reasoning patterns *robustly*, even when they are relatively rare (e.g., 4-6 step 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, 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 (Trivedi et al., 2022; Khot et al., 2022a).

Building upon these three strands, we introduce TEABREAC,[2] a teaching dataset that includes carefully constructed synthetic contexts for a broad set of real multi-step 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. Our approach relies on treating a question decomposition as an unambiguous typed program that can be used to generate a synthetic context and can be executed to provide an answer. To this end, we first turn natural language QDMRs into a precise typed *program*. We then construct a synthetic context by asserting a set of facts that relate to various parts of the multi-step 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. The resulting contexts are hard to cheat on and thereby force 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[3] large language models (LMs) on TEABREAC before fine-tuning on target multi-step 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)), NumGLUE (Mishra et al., 2022) by up to 13 F1 points, as well as on two contrastive evaluation sets of DROP by 5-8 points. Furthermore, even if we start with numerate LMs already pretrained on similar past work (Geva et al., 2020; Yang et al., 2021; Yoran et al., 2022; Pi et al., 2022), TEABREAC provides further improvement by up to 11 F1 points. Interestingly, TEABREAC is substantially more beneficial for more complex questions (those with more reasoning steps), improving the T5-Large model by about 20 F1 points on questions with 5 or more steps. More generally, we expect TEABREAC to be most valuable for datasets that require complex aggregation operations and their diverse compositions.

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 multi-step reasoning patterns and (b) leveraging existing QDMR annotations to carefully construct contexts that require true multi-step 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 numerate LMs much more effective and robust at multi-step reasoning, especially for more complex questions.

---

[2]TEABREAC = "**Te**aching **B**road **Rea**soning skills via decomposition-guided **C**ontexts"; pronounced "Tea Break".

[3]We use the word pretraining to mean fine-tuning on our generated QA data before (hence *pre*training) fine-tuning on the target dataset.

2 Related Work

Question Decompositions have been used to build stronger models (Talmor and Berant, 2018; Min et al., 2019b; Khot et al., 2021) and challenge evaluation sets by modifying the questions (Geva et al., 2022). In contrast, our goal in this work is to use decompositions to teach broad multi-step reasoning skills to any text-to-text model by creating challenging contexts for real questions.

Building synthetic datasets to teach requisite skills has been considered in prior work, but limited to only numeric reasoning skills (Geva et al., 2020; Yang et al., 2021) or few templated multi-step reasoning patterns (Yoran et al., 2022; Pan et al., 2021). Even pretraining on program executions (arithmetic, logic-based, and SQL-based) has been shown to help on multi-step QA tasks (Pi et al., 2022). In this work, we use real questions from a wide variety of datasets and show larger gains than these prior models. We even improve these prior models by fine-tuning on our dataset.

We create more robust models by teaching reasoning skills via a dataset carefully designed to avoid shortcuts. Past work often focuses on identifying lack of robustness via analysis (Min et al., 2019a; Trivedi et al., 2020) or challenge evaluation sets (Jiang and Bansal, 2019; Geva et al., 2022).

Lastly, we define new conditions for constructing contexts for real questions with minimal reasoning shortcuts. This differs from prior work that only provides conditions to measure reasoning shortcuts in *existing* datasets (Trivedi et al., 2020). The "MuSiQue condition" of Trivedi et al. (2022) targets the construction of *new* non-cheatable multi-step datasets. We enforce this condition in TEABREAC and introduce two additional ones that are especially pertinent to our construction. Appendix A includes additional discussion.

3 Teaching Broad-Coverage Reasoning Skills in a Robust Fashion

Multi-step 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 way to surface the reasoning needed for answering these questions is to look at their *decomposition* into smaller reasoning steps. E.g., 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 for 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 multi-step QA 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; Chen and Durrett, 2019; 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 it that is hard to cheat, 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 dataset, TEABREAC, by using real multi-step questions (from existing datasets) and their decompositions (available in the form of QDMRs), and carefully building synthetic contexts.

QDMR or Question Decomposition Meaning Representation (Wolfson et al., 2020) is a common way to represent the reasoning in many types of multi-step questions as a structured decomposition graph. QDMR has standardized operators (represented as nodes) such as `select`, `project`, `group`, 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 multi-step QA datasets.

Briefly, our method involves the following main steps; these are described in more detail in §4.

Making QDMRs more precise. To create QA instances that teach the precise reasoning in QDMRs, 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 inputs/outputs. Since this is crucial for our approach, we convert QDMRs into formal programs with over 44 executable primitive operations along with their input/output types (§ 4.1).

Teaching robust compositional skills. Past work has shown that compositional questions don't necessitate multi-step reasoning as datasets often have reasoning shortcuts (Min et al., 2019a; Chen
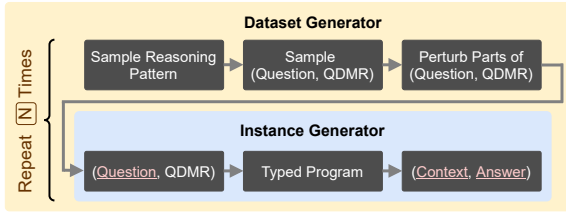
Figure 2: Schematic of TEABREAC construction.

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 (§ 4.2). 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 (§ 4.2).

Teaching a broad range of reasoning primitives. In addition to our process of constructing a pre-training 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 (§ 4.3).

4 TEABREAC Dataset Construction

The overview of TEABREAC construction pipeline is shown in Fig. 2. We discuss the QA instance generator in § 4.1 and the dataset generator in § 4.2.

4.1 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 RC QA instance. This conversion happens in two steps: (i) QDMR to Typed Program, (ii) Typed Program to Context and Answer.

4.1.1 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 generate C and A, we must be able to create facts corresponding to steps in the QDMR reasoning graph (i.e., ground the QDMR predicates[4]) and compute the final answer by stepping through it.

To achieve this, we need a formal representation (**Program**) that captures the precise reasoning implied by D, and that can be executed step-by-step (e.g., in a programming language like 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 make execute it.

To convert a QDMR D into a Program P, we define a set of python functions (primitives)[5] like `select`, `filter`, `grouped_count`, etc, and parse QDMRs into these functions using rules and heuristics. An example conversion is shown in Fig. 3.
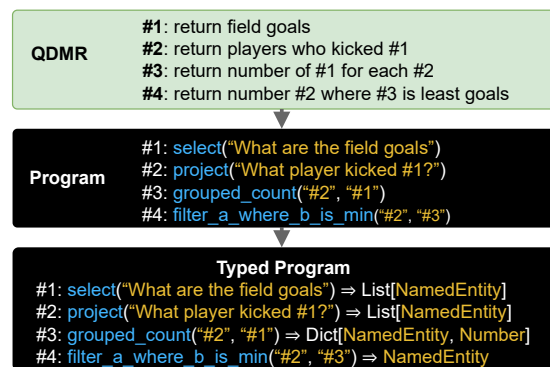
Figure 3: Example conversion of a QDMR decomposition (top) to a Typed Program (bottom).

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 them (`select`,

---

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

[5]We have 44 primitives operating over various types (number, date, named entity) and structures (scalar, list, dictionary) of inputs and outputs. The full list is given in App. I.

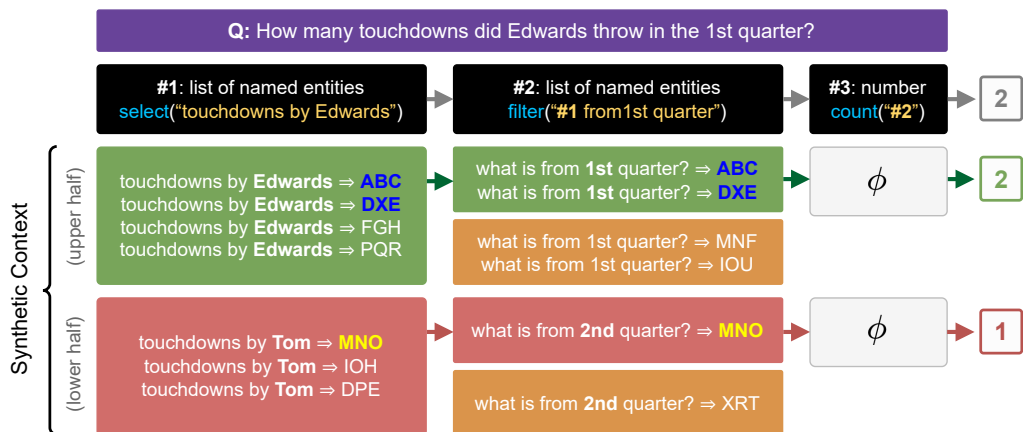Figure 4: A simplified example of a QA instance in TEABREAC, with a (simplified) real question from the DROP dataset and the synthetic context we construct for it using the question's 3-step decomposition. Statements in red, yellow and green form the synthetic context. The instance satisfies desirable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

project, filter), it's under-defined. E.g., select("number of soldiers in USA") should output a number, select("when did India get independence") should output a date, and select("countries surrounding India") should output a list of named entities. For such primitives, we use heuristic rules and type propagation on the global structure of P to infer expected types and structures of output. We call the program having type information for each step a **Typed Program** $\tilde{P}$, an example of which is shown in Fig. 3.[6]

4.1.2 Synthetic Context + Answer:

Next, we generate C and A from the typed program $\tilde{P}$. We generate C by grounding the **predicates** derived from the QDMR D with random **entities**. Fig. 4 shows an example of C for a program with three steps. **Predicates**: The predicates that need to be grounded belong to four primitives, i.e., select, project, filter, boolean. Example in Fig. 4 uses select and filter. Examples involving project and boolean are shown in App. H. **Entities:** The grounded entities are of 3 types: number, date, or named entity[7]. 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 with a number.

Minimizing reasoning shortcuts. Naively creating C using QDMR D can introduce shortcuts that models can exploit and bypass 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 the steps can be ignored. To ensure models learn the intended reasoning, our goal is to create C such that one can't 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 *"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 can't narrow down the answer to step #2 without knowing step #1's answer. We ensure this property for different operators differently. E.g., for filter, we ensure the answer 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 steps can't be the same, as otherwise the reasoning in that step can be bypassed. E.g., in Fig 4, step #2 is asking *"which of the touchdowns by Ed-*

---

[6]Since these programs are more precise and executable, future work may also use them to design better explicit multi-step reasoning systems (Khot et al., 2021).

[7]Numbers are in 0 to 1 million, dates are from year 1100 to 2022, and named entities are any sequence of 3 letters.

ward are from the 1st quarter". There are many *"touchdowns by Edward"*, but only some of them are *"from the 1st quarter"* (indicated in blue). So, ignoring step #2 (i.e., treating it as a no-op) would result in an incorrect answer being used for subsequent steps. We ensure this property for different operators differently. 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** ensure step-by-step execution will lead to the gold answer, but 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 have a third property:

P3: Context also supports a different answer to a contrastive question. Just as we generate facts for the gold chain of reasoning (upper half in Fig. 4), we also generate facts for distractor chain (lower half in Fig. 4), using potentially perturbed predicates (e.g., Edward ⇒ 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, different from the gold answer 2 in Fig. 4. To perturb predicates, 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 an entity, 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.

We note that past work of Trivedi et al. (2022) has also considered similar properties to create hard-to-cheat multi-step QA datasets. Our P1 is similar to the first part of their MuSiQue condition (the 2nd part isn't needed here as artificial entities make it impossible to ignore the context). Our P2 is new and especially pertinent to TEABREAC because of its list-based filter operations. Our P3 is also new and results in stronger question dependence than MuSiQue because of the emphasis on a minimally contrastive reasoning chain (as opposed to *any* additional reasoning chain which a context in MuSiQue often also supports).

To construct QA instances with properties P1-P3, we iterate through the program steps maintaining the step-wise answers and distractors for gold reasoning chain (upper half of Fig. 4) and the distractor reasoning chain (lower half of Fig. 4) respectively. For steps containing grounding predicates (`select`, `filter`, `project`, `boolean`), we ground the predicate with random entities of appropriate type and cardinality as defined by typed program. While doing such groundings we make sure the aforementioned properties satisfy. The final step answer is the answer A for the QA instance. The detailed description and pseudo-code to generate QA instances is given in App. B.

4.2 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 in the program. E.g., program in Fig. 4 has 3 steps having `select`, `filter` and `count` primitives, so the reasoning pattern is "`select filter count`".

Generating instances uniformly from such QDMRs would end up skewing the distribution of questions towards the popular patterns and result in the model overfitting to these patterns. To fix this, our dataset generator: (i) samples a reasoning pattern, (ii) samples a question-QDMR pair from that reasoning pattern, (iii) possibly perturbs question entities (named entities, dates, numbers, ordinals) with a closely similar entity of the same type,[8] and (iv) invokes the instance generator. The resulting training dataset has about 900 reasoning patterns with the top 10 common patterns having only 4% of examples (compared to 70% had we not done such balancing).

4.3 Additional QA Instances for Primitives

We also generate instances to teach 44 individual primitives, using simple templates similar to Geva et al. (2020). E.g., for primitive `filter_a_where_b_is_compared_to`, a question could be "Entities that have value larger than

---

[8]e.g., Edward ⇒ 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 §4.1.

948768.92?" and context could be "Entity AFE has value 871781. Entity RQX has value 989,517.24." resulting in the answer ['RQX']. App. I gives example instances for all the primitives. Each primitive has 30K training and 1K development instances.

4.4 Final Dataset

Final TEABREAC dataset has 525K and 15K train and development multi-step QA instances respectively, and has about 900 reasoning patterns. To create it we use publicly available QDMRs from QA and semantic parsing datasets, DROP (Dua et al., 2019), ComplexWebQuestions (Talmor and Berant, 2018), HotpotQA (Yang et al., 2018), SPI-DER (Yu et al., 2018), ComQA (Abujabal et al., 2019), ATIS (Price, 1990). We use both `low` and `high` level QDMRs limited to 2-6 reasoning steps.

5 Experiments

To test the effectiveness of TEABREAC pretraining, we compare models directly fine-tuned on target datasets with models first pretrained on TEABREAC[9] before fine-tuning.

Datasets. We evaluate **in-distribution performance** using DROP (Dua et al., 2019), TAT-QA (Zhu et al., 2021), IIRC (Ferguson et al., 2020), and NumGLUE (Mishra et al., 2022). For IIRC, we consider two settings: IIRC-G uses only gold supporting sentences as context while IIRC-R uses paragraphs obtained using a retrieval marginalization method (Ni et al., 2021). We evaluate **robustness** using the DROP contrast set (Gardner et al., 2020) and the DROP BPB contrast set (Geva et al., 2022)[10]. To do this, we directly evaluate DROP fine-tuned models on contrast sets.

Models. We evaluate TEABREAC pretraining on two kinds of (language) models or LMs. For **Plain LMs**, we use T5-Large (Raffel et al., 2020) and Bart-Large (Lewis et al., 2020). For **Numerate LMs**—those pretrained to perform numeric reasoning via different approaches—we use NT5 (Yang et al., 2021) based on T5-Small,[11] PReasM (Yoran et al., 2022) based on T5-Large, and POET (Pi et al., 2022) based on BART-Large. Henceforth,

we will use suffixes "-S", "-L", and "-3B" to refer to the model size.

We use author-provided checkpoints as our initial models and then fine-tune on the target datasets. Following NT5 and POET, we use character tokenization in all considered models during the fine-tuning stage. In some cases, prior work has also performed similar experiments (with different implementations and hyper-parameters) that we report in App. C for completeness.[12] Our models are implemented using PyTorch (Paszke et al., 2019), Huggingface Transformers (Wolf et al., 2019), and AllenNLP (Gardner et al., 2017). §G includes implementation details and training hyperparameters.

5.1 Results

TEABREAC improves model performance

In-distribution evaluation in Table 1 compares performance on DROP, TAT-QA, IIRC-G, IIRC-R and NumGLUE. For all considered plain language models (Bart-L, T5-L, and T5-3B), TEABREAC pretraining results in substantial improvements across all datasets — 4-7 F1 points on DROP, 10-13 F1 points on TAT-QA, 10-11 F1 points on IIRC-G, 4-10 F1 points on IIRC-R, and 5-7 F1 points on NumGLUE. For numerate language models (NT5-S, PReasM-L, and POET-L), TEABREAC pretraining improves performance by 2-3 F1 points on DROP, 1-11 F1 points on TAT-QA, and 3-15 F1 points on NumGLUE. TEABREAC pretraining doesn't improve NT5-S performance on IIRC-G and IIRC-R, but it improves PReasM-L and POET-L performances on both datasets by 2-7 F1 points. The fact that TEABREAC further improves models already pretrained on previous synthetic datasets highlights its complementarity.

Furthermore, TEABREAC-pretrained T5-3B achieves new state-of-the-art performance relative to the best previously published results on IIRC-G, IIRC-R, and NumGLUE, reported in the last row of Table 1. Moreover, even the smaller TEABREAC-pretrained PReasM-L and POET-L models improve over previously published numbers on IIRC-G and NumGLUE respectively. On DROP and TAT-QA, specialized architectures (with special task-specific modules) developed for those datasets outperform TEABREAC-pretrained models.

Since numerate LMs are derived by pretraining plain LMs on respective synthetic datasets, we can

---

[9]Models work well on TEABREAC (see App. E).

[10]We use the human validated set. We also remove yes/no questions from it as DROP does not contain yes/no questions but TEABREAC does, and hence it unfairly favors TEABREAC pretrained models.

[11]NT5 is only available in small size.

[12]Models with TEABREAC pretraining outperform both our and previously reported fine-tuning implementations.

Table 1:

	Model	In-distribution Evaluation					Robustness Evaluation	
		DROP	TAT-QA	IIRC-G	IIRC-R	NumGLUE	DROP-CS	DROP-BPB
Plain LMs	Bart-L	72.3 | 73.3	44.8 | 43.9	66.9 | 65.0	44.8 | 41.7	46.0 | 41.9	53.7	51.5
	+ TEABREAC ☕	81.3 | 80.7	54.2 | 53.7	76.2 | 75.3	48.5 | 45.6	52.5 | 49.1	61.8	59.3
	T5-L	76.1 | 77.1	47.2 | 46.3	68.0 | 63.6	45.4 | 38.9	49.7 | 42.9	53.4	56.4
	+ TEABREAC ☕	81.4 | 81.1	58.3 | 56.9	72.9 | 72.8	46.1 | 45.7	53.3 | 49.8	60.1	63.2
	T5-3B	82.0 | 82.1	49.8 | 51.1	70.9 | 68.2	46.4 | 40.9	54.9 | 49.7	61.8	63.6
	+ TEABREAC ☕	86.7 | 86.5	65.5 | 63.8	**78.6** | **79.5**	**52.5** | **51.0**	**57.3** | **54.3**	**66.8**	**69.7**
Numerate LMs	NT5-S	72.7 | 73.0	51.9 | 51.9	71.3 | 71.4	45.2 | 44.3	37.0 | 32.7	46.4	51.8
	+ TEABREAC ☕	75.1 | 75.3	53.4 | 52.8	70.4 | 70.3	44.9 | 44.2	50.7 | 47.5	52.9	54.2
	PReasM-L	80.0 | 80.2	48.7 | 49.7	74.5 | 73.3	45.5 | 40.9	52.3 | 46.4	57.3	56.1
	+ TEABREAC ☕	83.2 | 83.4	61.7 | 60.4	77.2 | 77.9	50.5 | 47.6	53.1 | 49.2	60.8	64.4
	POET-L	79.6 | 79.4	52.8 | 53.1	71.8 | 73.8	47.5 | 44.3	50.7 | 45.5	58.3	55.6
	+ TEABREAC ☕	82.2 | 82.1	55.6 | 54.1	76.8 | 76.0	49.1 | 46.6	53.4 | 50.2	64.0	60.7
	Best Published	87.5 | 87.8[a]	[b]81.3 | 78.0[b]	77.4 | 75.0[c]	50.6 | 50.5[d]	n / a | 48.8[e]	54.2[f]	65.9[g]

Table 1: F1 scores of in-distribution and robustness evaluation of language models (LMs) with and without ☕ TEABREAC pretraining on dev and test sets. Pretraining LMs on TEABREAC improves their in-distribution performance and robustness across multiple QA datasets, for both plain and numerate LMs. In-distribution evaluation scores are (dev | test) scores. Robustness evaluations are on test-only contrast sets. The suffixes '-3B', '-L' and '-S' refer to model sizes 3B, large and small, respectively. Green (underlined) indicates TEABREAC pretraining improves the underlying model's performance, while red (not underlined) indicates it does not. **Bold** indicates that the TEABREAC-pretrained model sets a new state of the art among published models. EM scores are provided in Appendix F. a : Zhou et al. (2022), b : Zhou et al. (2022), c : Yoran et al. (2022), d : Ni et al. (2021), e : Mishra et al. (2022), f : Gardner et al. (2020), g : Geva et al. (2022).

also directly compare such pretraining approaches with TEABREAC pretraining. From Table 1, we can also see that the T5-L + TEABREAC model is better than the PReasM-L model (T5-L pretrained on PReasM data), and the Bart-L + TEABREAC model is better than the POET-L model (Bart-L pretrained on POET data). See App. D for additional comparisons.

TEABREAC improves model robustness

We evaluate the robustness in Table 1 by comparing performance on the DROP contrast set and the DROP BPB set. For all plain language models, T5-L, T5-3B and Bart-L, TEABREAC pretraining shows substantial improvements in robustness — 5-8 F1 points improvements on DROP contrast set and on DROP BPB set. For numerate LMs, NT5-S, PReasM-L and POET-L, TEABREAC pretraining results in 4-7 F1 points of improvement on DROP contrast set and 2-8 points of improvement on DROP BPB set.

☕ improves more on more complex questions

We further investigate how the improvements provided by TEABREAC vary based on the complexity or number of steps of the question. To obtain the

number of reasoning steps, we use our programs.[13] But since QDMRs, and as a result programs, are not available for all the questions, we use the number of reasoning steps in predicted programs (using a T5-Large model trained on the BREAK dataset followed by conversion into our typed programs).

Figure 5 compares the performance of TEABREAC pretraining on questions with increasing (estimated) number of steps. While the T5-L baseline model drops significantly from 79 to 58, T5-L with TEABREAC pretraining stays mostly invariant to the number of steps. We thus observe a significantly larger improvement for more complex questions, where the original T5-L model struggles (e.g., 20 points gain on 4+ steps vs. 5 points gain on average). Similarly, for the numeracy-aware language model PReasM-Large, we see more improvement on more complex questions (e.g., 9-10 points on 4+ steps, 3.2 points on average). We see similar trends for the other models as well.

We also observe that more complex questions are much less frequent in the DROP development set (e.g., 4+ steps constitute only 25%). This makes our large gains on more complex questions not

---

[13]Our programs may differ in the number of steps than the source QDMR due to additional normalization and processing.
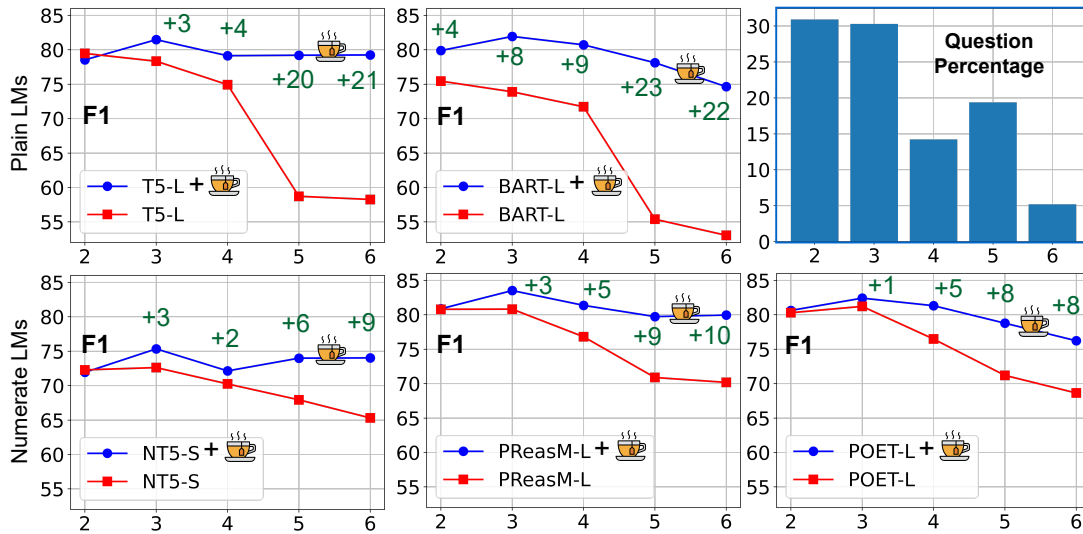
Figure 5: F1 scores for plain and numerate LMs with and without ☕ TEABREAC pretraining on DROP across varying numbers of steps, as determined by our programs. TEABREAC pretraining helps more on more complex questions. The effect is more prominent on plain LMs like T5-L than on numerate LMs like PReasM-L. (Top Right) Histogram of percentage of questions for each step count. Because more complex questions are less frequent, improvements by TEABREAC pretraining don't show up as well on the average metric for the entire dataset.

quite visible in the aggregate metric (Table 1).

TEABREAC Ablations

To assess the contribution of various aspects of TEABREAC to the overall performance, we perform ablation experiments with T5-L on the DROP dataset. Fig. 6 shows the results for DROP contrast set and BPB set. Pretraining on just primitive QA instances helps by 0.5-2.3 points, which further improves by 2.7-3.5 points when adding multi-step QA instances without QDMR-balancing (§ 4.2). Finally, if we add multi-step instances with QDMR-balancing instead, we get an additional 1.7-2.8 points of improvement. DROP development set has similar trends but with lower absolute differences, potentially due to shortcuts (see App. F).

Figure 6: ☕ TEABREAC Ablations: All aspects of TEABREAC pretraining data contribute to the overall performance: (i) primitive QA instances (ii) multi-step QA instances (iii) balancing of QDMR distribution.

6 Conclusions

Despite large LMs' impressive reading abilities and the availability of large scale multi-step QA datasets requiring a rich set of reasoning skills, LM-based QA models do not reliably learn to use such 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 from synthetic data to ac-

tual QA datasets add to the growing line of work that shows synthetic datasets can in fact be used to inject useful skills are valuable for real, natural language tasks. Given the artifact issues in real datasets (specifically, in their contexts) and the difficulty in controlling for them via perturbations, we present a viable alternative: leveraging existing multi-step questions for their broad reasoning patterns but using synthetic contexts for carefully constructing teaching datasets, where models can learn the *right way* to reason.

Ethical Considerations

The source dataset that TEABREAC is created from, i.e., BREAK, is publicly available with the MIT license which allows us to modify and release the dataset. TEABREAC models and datasets are released under the CC BY 4.0 License.[14]

Since TEABREAC uses questions and decompositions from existing datasets, it may also inherit the social biases present in these underlying datasets. We haven't taken any explicit steps to remove such potential biases as it's not in the scope of this work. But we advise the users of the TEABREAC dataset and models to take appropriate caution if deploying them in any real user-facing application.

Limitations

We proposed a pretraining approach to teach a broad range of multi-step reasoning skills to language models. Even though such pretraining doesn't have to be repeated for each target dataset, there is a significant computational cost to pretraining. E.g., our T5-Large pretraining takes 5 days on a RTX A6000 GPU. This is precisely the reason why we haven't conducted experiments with even larger models such as T5-11B. Identifying more compute-efficient ways to teach models such skills remains an interesting open problem.

In general, we expect TEABREAC pretraining to improve downstream performance on the datasets that require complex aggregation operations and diverse compositions of them. We have shown the effectiveness of TEABREAC pretraining on several multi-step QA datasets which fit this criteria. However, this is not the case for other multi-step QA datasets like QASC, HotpotQA, 2WikiMultihopQA, and MuSiQue, which involve simpler compositions. TEABREAC pretraining thus may not lead to similar gains on these datasets. More broadly, multi-step QA datasets we have considered form only a small subset of the vast number of QA and NLU tasks the NLP community is interested in. It's possible that TEABREAC pretraining is unhelpful and even harmful to the performance of LMs on these other tasks where our learned multi-step skills are not as relevant, such as commonsense understanding.

The skills taught in TEABREAC are limited by the skills captured (or capturable) by QDMRs. While expanding the scope of QDMR operators

and the datasets annotated with them can automatically expand the scope of TEABREAC, the current approach is still limited to datasets where one can easily define and obtain QDMRs.

Lastly, while TEABREAC enables the teaching of reasoning skills to any text-to-text model, these black-box models don't provide explanations, making it hard to analyze their underlying reasoning. Hence, we are unable to check whether models trained on it are necessarily performing the required multi-step reasoning. We only provide indirect empirical evidence by evaluations on contrast sets.

Acknowledgments

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.

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*.

---

[14]https://creativecommons.org/licenses/by/4.0

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*.

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

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. 2022a. Hey AI, can you solve complex tasks by talking to agents? In *Findings of ACL*, pages 1808–1823.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2022b. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406*.

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*.

Swaroop Mishra, Arindam Mitra, Neeraj Varshney, Bhavdeep Sachdeva, Peter Clark, Chitta Baral, and Ashwin Kalyan. 2022. NumGLUE: A suite of fundamental yet challenging mathematical reasoning tasks. In *ACL*, pages 3505–3523, Dublin, Ireland.

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.

Pruthvi Patel, Swaroop Mishra, Mihir Parmar, and Chitta Baral. 2022. Is a question decomposition unit all we need? *arXiv preprint arXiv:2205.12538*.

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*.

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 *ICML*, pages 4596–4604. PMLR.

Alon Talmor and Jonathan Berant. 2018. The web as a knowledge-base for answering complex questions. In *NAACL*.

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: Multi-hop 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.

Yongwei Zhou, Junwei Bao, Chaoqun Duan, Haipeng Sun, Jiahui Liang, Yifan Wang, Jing Zhao, Youzheng Wu, Xiaodong He, and Tiejun Zhao. 2022. OPERA: Operation-pivoted discrete reasoning over text. In *NAACL*, pages 1655–1666, Seattle, United States.

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 Related Work

Question Decomposition. Several recent multi-step 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., 2022a). These works have enabled the development of *explicit* multi-step reasoning systems that first decomposes a question into sub-questions, and answers the sub-questions step-by-step to arrive at the answer (Min et al., 2019b; Khot et al., 2021; Trivedi et al., 2022; Patel et al., 2022; Khot et al., 2022b). In contrast, our goal is to use decompositions to teach language models multi-step reasoning implicitly (within the model).

Since each dataset has its own decomposition format, they have led to narrow dataset-specific solutions. In contrast, the BREAK dataset (Wolfson et al., 2020) defined a standardized format for several QA datasets. So in this work, we use them to build a teaching dataset for broad reasoning skills.

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

The closest work to ours is the Break-Perturb-Build (BPB) dataset (Geva et al., 2022). BPB also uses QDMR but to create contrastive *questions* via small question perturbation (Kaushik et al., 2019; Gardner et al., 2020). Unlike us, they use the existing context with reasoning shortcuts that can be hard to eliminate with only question perturbation (e.g., no distractors). Additionally it is mainly used for evaluation (as we also do) and hasn't been shown to improve models by training on it.

Data Augmentation for QA. Several past works have used data augmentation via synthetic datasets to improve QA performance. Following 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 multi-step 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-step reasoning skills.

B Algorithm to Generate QA Instances

Algorithm 1 shows the pseudo-code for generating QA instances satisfying the three properties discussed in § 4. 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 stepwise answers for gold reasoning chain (e.g., green boxes in Fig. 4), and state.dis represents stepwise answers for distractor reasoning chain (e.g., red boxes in Fig. 4). These are initialized to $\emptyset$(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 properties 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 property P3 is satisfied.

The implementation of execute function is dependent on the program primitives (Table 7) and will be provided in the released code. 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

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 ≤ i ≤ 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
```

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.

C   Our Implementation vs Previously Reported Numbers

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. For a fair comparison of the fine-tuning experiments, we do the direct fine-tuning on the target datasets using our implementation instead of relying of previously reported numbers which may have other differences. Moreover, previously reported numbers are only sparsely available across the model-dataset pairs we consider, which is another reason to use our implementation. Table 2

shows results obtained by our implementation vs results reported by prior works (NT5 (Yang et al., 2021), PReasM (Yoran et al., 2022) and POET (Pi et al., 2022)), where available. Irrespective of implementation, models with TEABREAC outperform prior approaches.

Note that following Yang et al. (2021) and Pi et al. (2022), we employ character tokenization for numbers, but it wasn't employed by Yoran et al. (2022). Therefore, our results obtained by our implementation are significantly better than the ones reported in Yoran et al. (2022) for DROP, where numerical reasoning is crucial.

D   Direct Comparison of TEABREAC vs Previous Pretraining Methods

Table 1 shows that TEABREAC pretraining improves both plain and numerate LMs. However, since the numerate LMs (PReasM and POET) are derived by pretraining plain LMs on the respective synthetic datasets, we can also directly compare TEABREAC pretraining with PReasM and POET pretraining. Table 3 shows this comparison. The results are limited to T5-Large and Bart-Large as PReasM and POET checkpoints are not available in other sizes considered. We find that LM with TEABREAC pretraining outperforms LM with PReasM or POET pretraining.

E   Performance of LMs on TEABREAC

Since our goal is to teach models the reasoning skills in TEABREAC, we assess how well models do on the TEABREAC dataset. As shown in Table 4, models are able to learn both primitive and multi-step QA skills required in TEABREAC. On primitives instances models get 92-98 F1, and on multi-step instances, models get 84-88 F1. We show in our experiments 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 multi-step models.

F   Results in Exact Match (EM) metric

In addition to the F1 results reported in Table 1, we also report the corresponding EM numbers in Table 5. We see the same trends discussed in § 5.

	In-distribution Evaluation					Robustness Evaluation	
Model	DROP	TAT-QA	IIRC-G	IIRC-R	NumGLUE	DROP-CS	DROP-BPB
Plain LMs							
Bart-L (Pi et al. (2022))	69.2 | –	46.7 | –	—	—	—	—	—
Bart-L (our implemented)	72.3 | 73.3	44.8 | 43.9	66.9 | 65.0	44.8 | 41.7	46.0 | 41.9	53.7	51.5
+ TEABREAC ☕	81.3 | 80.7	54.2 | 53.7	76.2 | 75.3	48.5 | 45.6	52.5 | 49.1	61.8	59.3
T5-L (Yoran et al. (2022))	64.6 | 65.0	—	69.9 | 67.1	47.4 | 41.0	—	—	—
T5-L (our implemented)	76.1 | 77.1	47.2 | 46.3	68.0 | 63.6	45.4 | 38.9	49.7 | 42.9	53.4	56.4
+ TEABREAC ☕	81.4 | 81.1	58.3 | 56.9	72.9 | 72.8	46.1 | 45.7	53.3 | 49.8	60.1	63.2
Numerate LMs							
NT5-S (Yang et al. (2021))	70.3 | 70.8	—	—	—	—	—	—
NT5-S (our implemented)	72.7 | 73.0	51.9 | 51.9	71.3 | 71.4	45.2 | 44.3	37.0 | 32.7	46.4	51.8
+ TEABREAC ☕	75.1 | 75.3	53.4 | 52.8	70.4 | 70.3	44.9 | 44.2	50.7 | 47.5	52.9	54.2
PReasM-L (Yoran et al. (2022))	72.3 | 72.6	—	77.4 | 75.0	50.0 | 45.1	—	—	—
PReasM-L (our implemented)	80.0 | 80.2	48.7 | 49.7	74.5 | 73.3	45.5 | 40.9	52.3 | 46.4	57.3	56.1
+ TEABREAC ☕	83.2 | 83.4	61.7 | 60.4	77.2 | 77.9	50.5 | 47.6	53.1 | 49.2	60.8	64.4
POET-L (Pi et al. (2022))	80.6 | –	49.6 | –	—	—	—	—	—
POET-L (our implemented)	79.6 | 79.4	52.8 | 53.1	71.8 | 73.8	47.5 | 44.3	50.7 | 45.5	58.3	55.6
+ TEABREAC ☕	82.2 | 82.1	55.6 | 54.1	76.8 | 76.0	49.1 | 46.6	53.4 | 50.2	64.0	60.7

Table 2: Comparison of: (i) Results reported by prior works (NT5 (Yang et al., 2021), PReasM (Yoran et al., 2022) and POET (Pi et al., 2022)) where available (ii) Results obtained from our implementation (iii) Results obtained by our implementation with ☕ TEABREAC pretraining. Irrespective of implementation, models with TEABREAC outperform prior approaches. In-distribution evaluation scores are (dev | test) scores. Robustness evaluations are on test-only contrast sets. The scores are in terms of F1 metric. Green (underlined) indicates TEABREAC pretraining improves the underlying model's performance, while red (not underlined) indicates it does not.

	In-distribution Evaluation					Robustness Evaluation	
Model	DROP	TAT-QA	IIRC-G	IIRC-R	NumGLUE	DROP-CS	DROP-BPB
T5-L	76.1 | 77.1	47.2 | 46.3	68.0 | 63.6	45.4 | 38.9	49.7 | 42.9	53.4	56.4
PReasM (T5-L based)	80.0 | 80.2	48.7 | 49.7	74.5 | 73.3	45.5 | 40.9	52.3 | 46.4	57.3	56.1
T5-L + TEABREAC	81.4 | 81.1	58.3 | 56.9	72.9 | 72.8	46.1 | 45.7	53.3 | 49.8	60.1	63.2
Bart-L	72.3 | 73.3	44.8 | 43.9	66.9 | 65.0	44.8 | 41.7	46.0 | 41.9	53.7	51.5
POET-L (Bart-L based)	79.6 | 79.4	52.8 | 53.1	71.8 | 73.8	47.5 | 44.3	50.7 | 45.5	58.3	55.6
Bart-L + TEABREAC	81.3 | 80.7	54.2 | 53.7	76.2 | 75.3	48.5 | 45.6	52.5 | 49.1	61.8	59.3

Table 3: F1 scores of in-distribution and robustness evaluation of large-sized models with previous and our pretraining. Pretraining on TEABREAC leads to better results than pretraining on PReasM or POET. Green (underlined) indicates TEABREAC pretraining leads to better results, while red (not underlined) indicates PReasM or POET leads to better results.

Model	☕ Primitive	☕ Multi-step
Bart-L ☕	91.8	86.5
T5-L ☕	93.8	87.6
T5-3B ☕	94.2	88.9
NT5-S ☕	98.1	84.1
PReasM-L ☕	94.2	88.3
POET-L ☕	91.5	87.4

Table 4: F1 scores of models pretrained on TEABREAC on its Primitive and Multi-step dev sets. Models learn the skills required in TEABREAC during pretraining well, but achieving perfect score is challenging for vanilla LM-based neural models.
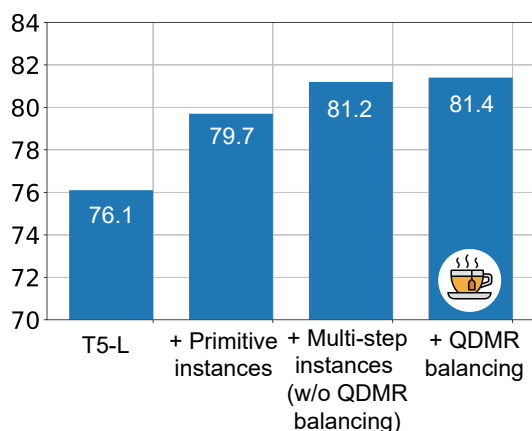

Figure 7: TEABREAC Ablations: All the three aspects of TEABREAC pretraining data contribute to overall performance: (i) primitive QA instances (ii) multi-step QA instances (iii) balancing of QDMRs to construct the multi-step QA dataset. The results are F1 scores on DROP dev set. The effect on DROP dev set is less prominent than in DROP CS and BPB sets, potentially due to shortcuts in DROP dev set.

TEABREAC Ablations on DROP dev set

TEABREAC ablation on DROP dev set is provided in Fig. 7.

G Implementation Details

We train all models on a RTX A6000 (48GB) GPU. We pretrain on TEABREAC by sampling a batch from multi-step and primitive synthetic instances in an alternating fashion. The hyperparameters for pretraining and fine-tuning are given in Table 6. The only hyperparameter we sweeped over is learning rate ($10^{-5}, 5 \times 10^{-5}, 10^{-4}, 5 \times 10^{-4}, 10^{-3}$). 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 exper-

iments (Shazeer and Stern, 2018). We selected training hyper-parameter (learning rate) for each baseline model and dataset, based on the validation set performance. Our fine-tuning experiments using models pretrained on TEABREAC use this identical learning rate.

H Examples of Multi-Step QA Instances

Example multi-step QA instances with project and boolean primitives are shown in Fig. 8.

I List of Primitives (Python Functions)

List of primitives (python functions) and a corresponding example is given in Table 7.

	Model	In-distribution Evaluation					Robustness Evaluation	
		DROP	TAT-QA	IIRC-G	IIRC-R	NumGLUE	DROP-CS	DROP-BPB
Plain LMs	Bart-L	69.2 | 70.0	37.4 | 35.8	62.4 | 60.5	41.7 | 39.1	44.5 | 40.4	47.0	46.7
	+ TEABREAC ☕	78.0 | 77.1	45.9 | 43.9	72.4 | 70.9	45.2 | 42.5	50.9 | 47.6	52.8	53.9
	T5-L	73.2 | 73.9	39.4 | 37.4	64.0 | 59.3	42.4 | 36.0	48.2 | 41.3	46.7	51.2
	+ TEABREAC ☕	78.2 | 77.8	50.4 | 47.8	69.1 | 69.0	43.1 | 42.7	51.7 | 48.3	52.7	57.3
	T5-3B	79.1 | 79.5	42.5 | 43.1	66.3 | 64.2	44.0 | 38.0	53.4 | 48.2	54.7	58.3
	+ TEABREAC ☕	83.9 | 83.8	58.2 | 55.8	74.5 | 75.4	49.0 | 48.4	55.8 | 52.8	58.6	64.4
Numerate LMs	NT5-S	69.2 | 69.4	44.2 | 42.3	66.6 | 66.9	41.9 | 41.7	34.2 | 29.2	38.8	46.3
	+ TEABREAC ☕	71.7 | 71.7	44.8 | 43.3	65.6 | 65.3	42.0 | 41.6	49.3 | 46.0	45.5	48.2
	PReasM-L	76.9 | 77.0	40.8 | 41.2	70.0 | 69.1	42.1 | 38.1	50.9 | 44.8	49.9	50.6
	+ TEABREAC ☕	80.1 | 80.1	54.5 | 51.6	73.0 | 72.9	47.3 | 44.6	51.5 | 47.7	53.6	59.0
	POET-L	76.6 | 76.3	45.6 | 44.6	67.6 | 69.7	44.4 | 41.7	49.2 | 44.0	51.2	50.9
	+ TEABREAC ☕	79.1 | 78.6	47.5 | 45.0	72.2 | 71.5	46.2 | 44.0	51.9 | 48.6	55.4	54.8

Table 5: EM scores of in-distribution and robustness evaluation of language models (LMs) with and without ☕ TEABREAC pretraining on dev and test sets. Pretraining LMs on TEABREAC improves their in-distribution performance and robustness across multiple QA datasets, for both plain and numerate LMs. In-distribution evaluation scores are (dev | test) scores. Robustness evaluations are on test-only contrast sets. The suffixes '-3B', '-L' and '-S' refer to model sizes 3B, large and small, respectively. Green (underlined) indicates TEABREAC pretraining improves the underlying model's performance, while red (not underlined) indicates it does not.
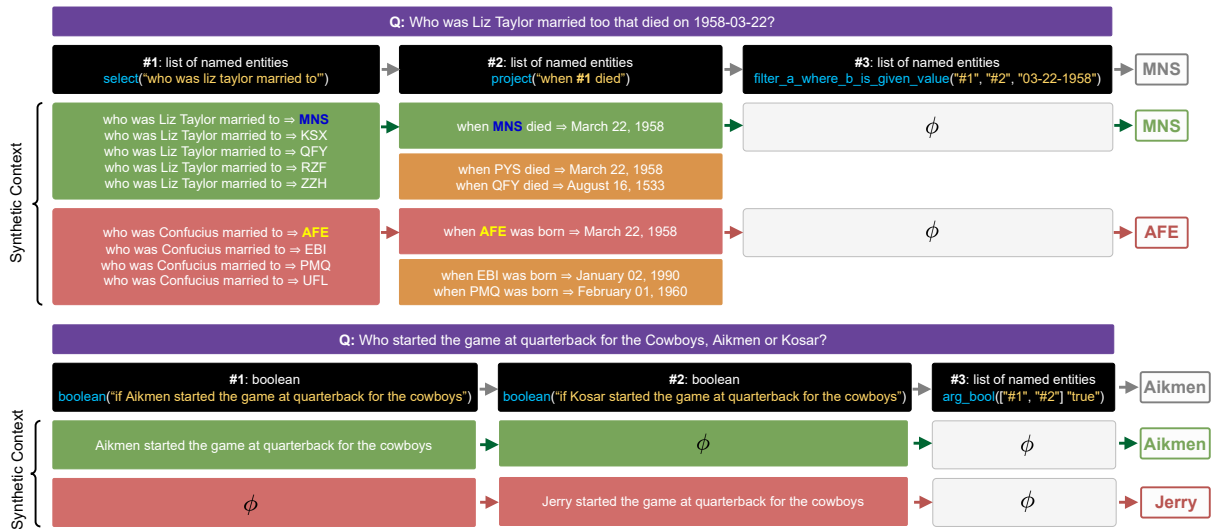

Figure 8: Synthetic reading comprehension QA instances involving project (top) and boolean (bottom) primitives.

Model	Dataset	LR	Epochs	BS
Bart-L	TEABREAC	10^{-5}	20	16
T5-L	TEABREAC	10^{-4}	20	8
T5-3B	TEABREAC	5×10^{-5}	10	8
NT5-S	TEABREAC	10^{-3}	20	32
PReasM-L	TEABREAC	5×10^{-5}	20	8
POET-L	TEABREAC	10^{-5}	20	16
Bart-L	DROP	10^{-5}	20	8
T5-L	DROP	10^{-4}	20	8
T5-3B	DROP	5×10^{-5}	10	8
NT5-S	DROP	10^{-3}	40	32
PReasM-L	DROP	5×10^{-5}	20	8
POET-L	DROP	10^{-5}	20	8
Bart-L	TAT-QA	10^{-5}	20	8
T5-L	TAT-QA	10^{-4}	20	8
T5-3B	TAT-QA	5×10^{-5}	10	8
NT5-S	TAT-QA	10^{-3}	40	32
PReasM-L	TAT-QA	5×10^{-5}	20	8
POET-L	TAT-QA	10^{-5}	20	8
Bart-L	IIRC	10^{-5}	20	8
T5-L	IIRC	10^{-4}	20	8
T5-3B	IIRC	5×10^{-5}	10	8
NT5-S	IIRC	10^{-3}	40	32
PReasM-L	IIRC	5×10^{-5}	20	8
POET-L	IIRC	10^{-5}	20	8
Bart-L	NumGLUE	10^{-5}	20	8
T5-L	NumGLUE	10^{-4}	20	8
T5-3B	NumGLUE	5×10^{-5}	10	8
NT5-S	NumGLUE	10^{-3}	40	32
PReasM-L	NumGLUE	5×10^{-5}	20	8
POET-L	NumGLUE	10^{-5}	20	8

Table 6: **Top:** Hyperparameters (HPs) for pretraining LMs on TEABREAC. For large and 3B sized models, each epoch constitutes 200000/batch-size steps. For the small sized model (NT5-S), each epoch constitutes 2000000/batch-size steps. For each step, we uniformly randomly sample a batch of TEABREAC multi-step instances or primitive instances. We've used identical HPs for pretraining ablations discussed in § 5.1. **Bottom:** HPs for fine-tuning LMs on target datasets. We use the same HPs for fine-tuning LMs with or without TEABREAC pretraining. The HPs for IIRC-G and IIRC-R experiments are the same. LR is learning rate and BS is batch size.

Table 7: 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

Table 7 – *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]

Table 7 – *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]

Table 7 – *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

J   Examples of Instances for Individual Primitives

Examples of template based QA instances for teaching individual primitives are given in Table 8.

Table 8: 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']

Table 8 – *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']

Table 8 – *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']