Active Programming by Example with a Natural Language Prior

Ruiqi Zhong[*1] **Charlie Snell**[*1] **Dan Klein**[1] **Jason Eisner**[2]
[1] University of California, Berkeley [2] Microsoft Semantic Machines
{ruiqi-zhong, csnell22, klein}@berkeley.edu
jason.eisner@microsoft.com

Abstract

We introduce APEL, a new framework that enables non-programmers to indirectly annotate natural language utterances with executable meaning representations, such as SQL programs. Based on a natural language utterance, we first run a seed semantic parser to generate a prior over a list of candidate programs. To obtain information about which candidate is correct, we synthesize an input on which the more likely programs tend to produce different outputs, and ask an annotator which output is appropriate for the utterance. Hence, the annotator does not have to directly inspect the programs. To further reduce effort required from annotators, we aim to synthesize simple input databases that nonetheless have high information gain. With human annotators and Bayesian inference to handle annotation errors, we outperform Codex's top-1 performance (59%) and achieve the same accuracy as the original expert annotators (75%), by soliciting answers for each utterance on only 2 databases with an average of 9 records each. In contrast, it would be impractical to solicit outputs on the original 30K-record databases provided by SPIDER.

1 Introduction

Semantic parsing often aims to map a natural language utterance u to a program s, which can be executed on an input i (Zettlemoyer and Collins, 2007; Kushman and Barzilay, 2013). For example, for $u =$ "*How old is the youngest person*," we can map to the SQL program $s =$ `SELECT MIN(AGE) FROM PEOPLE`, execute it on an input database i, and return the output answer $o \stackrel{\text{def}}{=} s(i)$[1] to the user. However, it is challenging to collect many (u, s) pairs for training a semantic parser, since annotating u with s requires an expert in the target programming language.

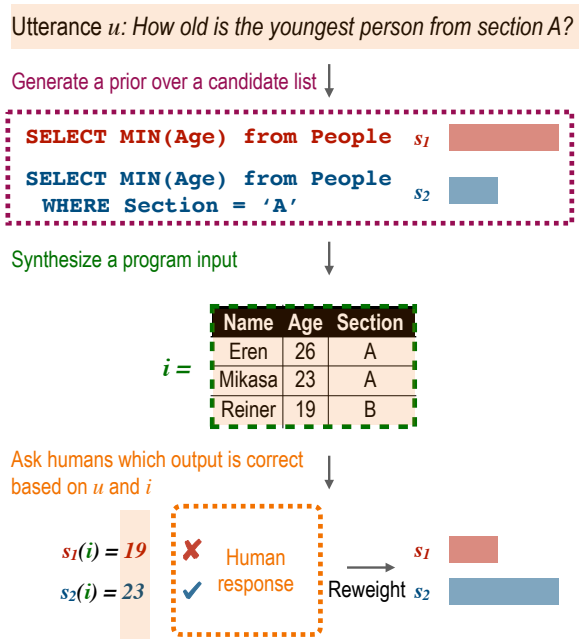

Figure 1: The outline of APEL, where the orange background indicates what the annotators can see. We upweight the program candidate consistent with the annotator's response. Figure 3 illustrates the criteria we used to synthesize the program input.

The programming by example (PBE) framework (Lavrac and Dzeroski, 1994) opens up a possibility: even though non-programmers are incapable of annotating u by directly writing the corresponding program s, they can produce example[2] input-output pairs (i, o) showing the intended effect of u. Then a program synthesis algorithm can guess the intended program s based on the constraint that $s(i) = o$ for every example pair. However, to have a good chance of pinning down the correct program for each utterance, annotators might need to produce a large set of examples, some of which will probably be redundant. In this paper, we attempt to elicit just

[1] $s(i)$ denotes the value returned by program s on input i.

[2] Our paper uses "example" to mean an input-output pair that shows how a specific program should behave, rather than an utterance-program pair that shows how the semantic parser should behave.

the necessary examples. We call this *active PBE*, by analogy with active learning.

We introduce **APEL**, a new framework that enables non-programmers to indirectly annotate each utterance with a program by answering questions about the desired behavior of the program. We first run a seed semantic parser on u to generate a list of candidate programs s ranked with probability scores $p(s)$ (Figure 1 top).[3] We then ask the annotators for information that helps us decide which candidate is correct. The annotators' answers are (possibly noisy) signals that allow us to better guess the correct candidate, or more generally to refine the probability distribution p, which could then be used to retrain the model or train a new model (Stiennon et al., 2020; Ouyang et al., 2022).

Utterance u: Find the first name of students who have both cat and dog pets.

s_1=
```
SELECT t1.fname
FROM student AS t1
JOIN has_pet AS t2 ON t1.stuid = t2.stuid
JOIN pets AS t3 ON t3.petid = t2.petid
WHERE t3.pettype = 'cat' INTERSECT
  SELECT t1.fname
  FROM student AS t1
  JOIN has_pet AS t2 ON t1.stuid = t2.stuid
  JOIN pets AS t3 ON t3.petid = t2.petid WHERE t3.pettype = 'dog'
```

s_2=
```
SELECT fname
FROM Student
WHERE StuID IN
    (SELECT T1.stuid
     FROM student AS T1
     JOIN has_pet AS T2 ON T1.stuid = T2.stuid
     JOIN pets AS T3 ON T3.petid = T2.petid
     WHERE T3.pettype = 'cat' INTERSECT
       SELECT T1.stuid
       FROM student AS T1
       JOIN has_pet AS T2 ON T1.stuid = T2.stuid
       JOIN pets AS T3 ON T3.petid = T2.petid WHERE T3.pettype = 'dog')
```

Figure 2: It is hard to spot and understand the differences between the complex programs s_1 and s_2. The error of s_1 is subtle: it finds the first names of the students who have dogs and the first names of the students who have cats, and intersects these two sets of first names. APEL can reveal this error by asking about a tiny student database i in which Alice Jones owns a dog while Alice Smith owns a cat. As no student named Alice owns both, the annotator will choose an output o that does not include `Alice`, implying that s_1 is incorrect. The annotator does not have to inspect s_1 or s_2 directly.

We focus on these indirect signals because, as illustrated in Figure 2, it is challenging even for programmers to directly choose the correct candidate. Instead, we synthesize a program input i, execute the candidates on i, ask a human which output is the intended response to utterance u given input i, and upweight the programs in p that yield this correct output (Figure 1 bottom). Since one input is not always enough to pin down the correct candidate, we iteratively reduce the entropy by repeating the process with new inputs.

[3] We assume that the list contains a correct program, which is often true. §9 discusses this assumption further.

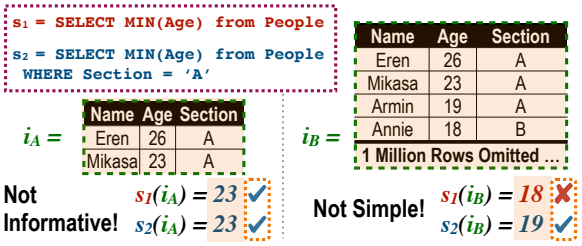

Figure 3: Criteria of informativity and simplicity guide our search for a useful program input. i_A is not informative since knowing the correct output does not tell us which s is correct. i_B is not simple since the input is too large to find the correct output for humans. Figure 1 shows an ideal input.

The total annotation effort depends on the number of questions we ask and also how hard it is to answer each question. Thus, we synthesize an i that 1) maximizes the expected information gain of knowing the annotator's answer, subject to the constraint that 2) it is simple enough that the annotator can easily determine the correct output (Figure 3).

Our method in effect applies active learning for each utterance u. To learn the function (program) s, we maintain a belief over the hypothesis space (candidate programs), and sequentially request the input labels that we expect will best refine our belief. Our main novelty is using active learning to bridge the semantic parsing and the PBE framework for program synthesis. Concretely, we leverage our prior knowledge of the desired program semantics, which we obtain from the natural language utterance u before we see any example (i, o) pairs. Therefore, we call our framework APEL—**A**ctive **P**rogramming by **E**xamples using a prior derived from a natural **L**anguage utterance.

As a case study, we apply APEL to indirectly annotate SPIDER (Yu et al., 2018), an English text-to-SQL dataset with >60 leaderboard submissions. Hence, s refers to a **S**QL program, i refers to an **i**nput database that can be queried by s, and p is the **p**rior over a list of SQL candidates, where Codex (Chen et al., 2021a) generates both the probabilities and the list. §3 proposes a heuristic to synthesize a small database i with high information gain.

We compare the highest-weighted program to the gold standard to evaluate our annotation framework. Using our synthesized input database i, a simulated perfect annotator with APEL can achieve 91% annotation accuracy on SPIDER by inspecting 2 databases per utterance with on average 6

records, while using the original sample database from SPIDER can only achieve 86% accuracy with on average 30K records (§5).

We then built an interface based on APEL and recruited 11 English-speaking non-experts to annotate a random subset of 240 utterances from the SPIDER development set, with each utterance examined by 2.5 humans on average (§6). Our system achieves the same accuracy as the original SPIDER annotation performed by database experts (75%), which significantly outperforms the top-1 accuracy of Codex (59%). §7 analyzes errors made by database experts without APEL and by our non-experts unfamiliar with databases.

Finally, §8 places APEL in the context of broader research trends. §9 ends by summarizing the prerequisites for applying it to other domains and future directions for improvement.

2 Framework

Using text-to-SQL as a case study, we aim to enable humans to indirectly annotate natural language utterances with SQL programs as accurately as possible. We take a Bayesian approach.

2.1 Outline of APEL

Let u be a natural language utterance and c a database schema.[4] We want to synthesize a SQL program s that captures the meaning of u and works properly for *any* database with schema c.

We first feed c and u to a seed semantic parser (e.g. Codex) to generate a prior distribution p over SQL programs s. We then synthesize an input database i and ask the annotators to choose the correct output o given u and i. Conditioning on these annotations raises the posterior probabilities of the candidates s such that $s(i) = o$.

Since we ask the annotators to select the correct output of a program given an input, we call our questions "*o-selection questions*."[5] We generate an o-selection question by 1) synthesizing a database i consistent with the schema c, 2) finding the possible outputs by executing the candidates from p on the database i, and 3) displaying u, i, and the most probable outputs (up to $M = 6$ distinct outputs, in random order). The annotator generates a response r that selects the output he/she thinks is the correct output of u on i (or $r =$"none of them is correct").

Our prior distribution over s was $p(s \mid u)$, and our posterior distribution given the response r is

$$\begin{aligned} p(s \mid u, i, r) &\propto p(s \mid u)\, p(r \mid u, s, i) \\ &= p(s \mid u)\, p(r \mid s(i)) \end{aligned} \tag{1}$$

Here $p(r \mid s(i))$ is our annotator behavior model, which models the probability that the annotator would have responded with r if s were a correct implementation of u and therefore $s(i)$ were the correct output. Most simply, if we assume that the annotator always responds correctly, we can calculate the posterior by zeroing out all candidates s in the prior that are inconsistent with r (e.g., $s(i) \neq r$) and then renormalizing. In this case, any correct candidates are guaranteed to be upweighted.[6]

Figure 1 illustrates this pipeline. We can ask more o-selection questions to obtain further information about the correct candidate and improve the posterior. For each utterance u, we define

$$\begin{aligned} p_t(s) &\stackrel{\text{def}}{=} p(s \mid u, i_1, r_1, \ldots, i_t, r_t) \\ &\propto p_{t-1}(s)\, p(r_t \mid s(i_t)) \end{aligned} \tag{2}$$

to be the posterior after t rounds of interaction, with $p_0 \stackrel{\text{def}}{=} p(s \mid u)$ being the prior and p_T being our final estimate. In our present experiments, we treat each annotator independently, conditioning on previous responses from only the same annotator. We ask up to 3 consecutive questions of the annotator, stopping the interaction after 1 or 2 questions if some candidate s already has $p_t(s) > 0.9$, or if our heuristic (§3) fails to find an appropriate question.

To evaluate our framework, we output the most probable SQL candidate according to p_T as our annotation, and compare it to a gold standard.

Appendix A explains how the "soft annotations" provided by the full distribution p_T for each utterance u could be used to retrain the semantic parser $p(s \mid u)$ and also the full annotator behavior model $p(r \mid u, s, i)$. These improved models would feed back to improve the estimate of p_T (i.e., an EM algorithm), and APEL could also use them to better select future questions. Appendix B offers some additional extensions that would better choose who should annotate which utterance at each step.

[4] The schema c specifies the table and column names, along with constraints that the database must satisfy, such as value type, uniqueness, and foreign key constraints.

[5] In principle, active PBE could also accommodate other kinds of questions whose responses are correlated with s.

[6] Except in the case of a useless question with only one possible response, so that no candidates can be zeroed out.

2.2 Criteria for Synthesized Inputs

To generate an o-selection question on round t, APEL needs to synthesize an input database i_t that is both **informative** and **simple**.

Informative. Our belief as we enter round t is p_{t-1}. Once we observe the annotator's response r_t, we will be able to update it to p_t. This will achieve an **information gain** of $\mathrm{H}(p_{t-1}) - \mathrm{H}(p_t)$, where the Shannon entropy H of a distribution over programs s characterizes its remaining uncertainty about which program is correct.

However, p_t will depend not only on our choice of question i_t but also on the annotator's response r_t (equation (2)). We do not know r_t yet, but our current belief is that it will be distributed as

$$p_{t-1}(r_t) = \sum_s p_{t-1}(s)\, p(r_t \mid s(i_t)) \qquad (3)$$

So our *expected* **I**nformation **G**ain from asking i_t is

$$\mathrm{IG}_{p_{t-1}}(i_t) \stackrel{\text{def}}{=} \mathrm{H}(p_{t-1}) - \mathbb{E}_{r_t \sim p_{t-1}}[\mathrm{H}(p_t)] \qquad (4)$$

where the expectation is taken under distribution (3). This is high if the candidates s that are most plausible under p_{t-1} tend to return different outputs $s(i_t)$ and hence different annotator responses r_t, making r_t informative about s.[7] By contrast, i_A in Figure 3 left would yield an uninformative response (IG $= 0$). Expected information gain has long been used to choose useful questions for experimental design (Lindley, 1956), decision tree construction (Quinlan, 1986), and active learning of parametric functions (MacKay, 1992).

Simple. We would like to avoid presenting the annotator with complex inputs such as the large database i_B in Figure 3 right. The correct response might be informative, but determining it would require too much human effort. We crudely model the effort required for i_t as the number of records $|i_t|$.

The next section proposes a heuristic to synthesize a simple informative database i_t given schema c, a sample database with schema c, and a distribution p_{t-1} over SQL programs.

3 Optimizing the Input Database

We attempt to maximize the expected information gain IG over all databases that conform to schema c (footnote 4) and have at most $R = 15$ total records. Formally, we search for

$$i_t^* = \operatorname*{argmax}_{i_t : |i_t| \le R} \mathrm{IG}_{p'}(i_t). \qquad (5)$$

where p' is a truncation of p_{t-1} to just the top-16 SQL candidates for computational efficiency. When multiple databases have the same IG, we break ties by favoring the smallest database. Since t and p' are fixed during the optimization process, we will write $\mathrm{IG}(i)$ instead of $\mathrm{IG}_{p'}(i_t)$ for short.

Our method can be summarized as "fuzz-then-drop." Fuzzing (Miller et al., 1990) is an established practice in software testing, where an algorithm generates a large number of random program inputs to search for an input that satisfies a property or reveals a bug. In our case, we want to search for an input database that maximizes the information gain. Therefore, we first perform fuzzing by randomly generating a large number of large databases as in Zhong et al. (2020)—see Appendix C for further details—and keep the database i^0 that maximizes the expected information gain $\mathrm{IG}(i^0)$. We then iteratively drop records from i^0 to satisfy the simplicity criterion.

We use superscript ℓ to denote the iteration of dropping records. Starting from $\ell = 0$, we randomly drop 5% of the records from i^ℓ to obtain $i^{\ell+1}$. We repeat this up to 20 times, and keep the value of $i^{\ell+1}$ that maximizes $\mathrm{IG}(i^{\ell+1})$, stopping early if we find any $i^{\ell+1}$ with $\mathrm{IG}(i^{\ell+1}) > \mathrm{IG}(i^\ell)$. We then increment ℓ and repeat the procedure, thus generating $i^0, i^1, \ldots$ until we reach an empty database i^L after $L = \Theta(\log |i^0|)$ iterations.

Of the databases smaller than R that we encountered during these iterations, let $\hat{i}$ be the one with the highest IG:

$$\hat{i} \stackrel{\text{def}}{=} \operatorname*{argmax}_{i \in \{i^\ell : 1 \le \ell \le L\}, |i| \le R} \mathrm{IG}(i) \qquad (6)$$

Since our procedure is randomized, we repeat it 3 times, and let i^* be the $\hat{i}$ with the largest $\mathrm{IG}(\hat{i})$,[8] breaking ties as before in favor of smaller $\hat{i}$. Finally, we simplify i^* by dropping tables and columns that were not mentioned by any of the top-16 SQL candidates (those in p').

Our procedure of dropping records from a large informative database is heavily inspired by Miao

[7] Indeed, the IG (4) can be regarded as the conditional mutual information between the random variables r_t and s, given all previous questions and answers. Hence it is ≥ 0.

[8] We fail if $\mathrm{IG}(i^*) = 0$. For example, the SQL query `SELECT B FROM TABLE LIMIT 100` returns the first 100 values of column `B`. It cannot be distinguished from `SELECT B FROM TABLE` by any i whose `TABLE` has $\le$ 100 records.

et al. (2019), which, given a database i such that $s_1(i) \neq s_2(i)$, provably finds the smallest subset of records in i such that s_1 and s_2 return different outputs. However, their algorithm works only for a restricted family of SQL programs and cannot be adapted to optimize information gain. Our procedure does not provide any provable optimality guarantee, but is more flexible and practical.

In practice, simply applying the above algorithm can generate unnatural databases and lead to vacuous SQL execution, confusing the annotators. In Appendix C, we illustrate several typical confusions (Figure 5) and explain how we fix them.

4 Experimental Setup

We describe the dataset used to benchmark APEL (§4.1), how we obtain the prior over SQL candidates (§4.2), and our evaluation metrics (§4.3).

4.1 Dataset

We benchmarked APEL on the development set of SPIDER (Yu et al., 2018), an English text-to-SQL dataset with 1034 utterance-SQL pairs distributed under the CC BY-SA 4.0 License. The SPIDER development set is divided into 20 *domains*, where each domain uses the same database schema c and has a collection of utterance-SQL (u, s) pairs and a sample database. To make evaluation less noisy, we fix the sample database to conform to the database schema c and update the test suite correspondingly (see Appendix D).

We used half of the 1034 (u, s) pairs (the *validation split*) to manually develop our annotator interface (§6) and our fuzz-then-drop algorithm (§3). The validation split was also used to prompt Codex (§4.2). We used the remaining half (the *evaluation split*) to evaluate our system with simulated annotators (§5), and from these drew a random subset of 240 utterances[9] to evaluate our system with actual human annotators (§6). To make the latter evaluation less noisy, we corrected errors in 61 out of these 240 SQL annotations (§7.1). We also identified and fixed several issues with the SPIDER database schema and content (see Appendix D). The corresponding author of the SPIDER dataset endorses our corrections and database updates (T. Yu, p.c.). We will release our dataset split and corrected annotations for reproducibility.

[9]Balanced across domains as much as possible.

4.2 Obtaining the SQL Program Prior p_0

We generated a prior over SQL program candidates using the Codex (Chen et al., 2021b) language model with few-shot prompting. Given an utterance u with a schema c, we created the prompt (Figure 6) by concatenating a linearization of c with eight (u_k, s_k) pairs from the validation split[10] associated with the same schema c, and finally the utterance u itself. Some (u_k, s_k) pairs were chosen randomly while others were chosen because u_k has high TF-IDF similarity to u. We randomly sampled 200 prompts for u by choosing different (u_k, s_k) pairs, and for each prompt, we asked Codex to generate 20 completions (SQL programs) and filtered out non-executable candidates. Appendix E.1 includes the full details. We did not perform any task-specific engineering.

Since our ultimate goal is to generate semantically correct SQL programs, we merged semantically equivalent candidates by testing them on 1K random databases with code from Zhong et al. (2020) to prevent surface form competition (Holtzman et al., 2021). We define p_0 to be the empirical distribution of semantic equivalence classes in our samples; thus, each s in §2 represents an equivalence class rather than a single SQL program.

Treating the original SPIDER annotation as the ground truth, the top-1 accuracy of p_0 on the entire development set is 72% and the top-16 accuracy is 94%. More details are in Appendix E.2. These numbers are not comparable to prior works, which usually evaluate on unseen database domains in a zero-shot manner (harder than our setting) but do not require predicting string literals and `DISTINCT` keywords (which we need for execution).

4.3 Evaluation Metrics

We evaluate APEL by comparing the posterior 1-best SQL program to the gold annotation for each utterance. We decompose its errors into three categories. First, recall from §4.2 that we only consider 4K sample program candidates (some being duplicates), so the correct SQL program might not appear in our candidate list. Second, recall from §2.1 that we may stop asking questions before the correct candidate becomes the most probable one. Third, the annotators sometimes respond incorrectly.

To reflect these three types of error, we calculate

[10]Excluding pairs where s_k matches the correct answer s.

1) the *candidate ceiling*—whether *any* candidate is semantically correct; 2) the *interaction ceiling*—our 1-best posterior accuracy if we assume an oracle annotator who always responds correctly, and 3) the *annotation accuracy*—our 1-best posterior accuracy given the actual annotations we collected.

5 Simulated Evaluation

We benchmarked APEL on the entire evaluation split under the idealistic assumption that 1) the SQL annotation provided by SPIDER is always correct, and 2) our annotator always responds correctly by choosing the output of that SQL program.

The candidate ceiling is 96% and the interaction ceiling is 91%, which is in fact much higher than the current annotation accuracy in SPIDER, as we will see in §6. Using the databases i generated in §3, we only need to interact with our idealized annotator for 1.8 rounds on average, and the databases that we present contain only 5.52 records on average. Appendix G includes more detailed statistics.

What if we had instead asked the oracle annotator to evaluate each u on SPIDER's sample database i from u's domain? Those questions are less informative: using them lowers the interaction ceiling to 86%. They are also far less simple: their median size is 72 records (and their mean size is 33,295 records due to large outliers). Human annotators cannot feasibly evaluate an utterance on such large databases. This motivates APEL's construction of simple informative databases.

6 Human Evaluation

We built an interface designed to be user-friendly (§6.1) and used it ourselves to establish gold annotations for 240 utterances (§6.2). We then recruited 11 non-experts to annotate the same utterances (§6.3), aggregated their responses by learning a model of annotator behavior (§6.4), and benchmarked their performance with the newly established gold standard (§6.5).

6.1 Annotation Interface

Figure 4 shows our interface. As described in §2, the annotator is required to choose the correct output on the right of the screen, or report that none of the choices are correct based on the utterance u (top) and the database i (left). In general, an output may be a string, a number, or a table. To collect additional information for qualitative analysis, the annotator can also use an open-ended response

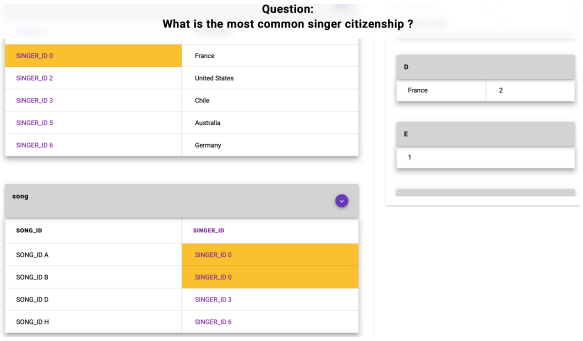

Figure 4: A screenshot of our annotation interface (§6.1). Appendix I and Figure 10 include more details.

field to optionally report that the question is ambiguous or confusing.[11] To make it easier to answer u based on i, we highlight all cells of i that have the same value as the cell that the cursor hovers on. Appendix I provides more details about our interface.

6.2 Expert Annotation

To establish a clean gold standard, two of the authors annotated all 240 utterances using our own APEL system. Whenever our two responses to a APEL question were different, we reached a consensus through discussion.

We closely examined the utterances where SPIDER's SQL annotation did not yield our consensus response on one or more questions, and corrected the SPIDER annotation if we felt that our responses were strictly better. To avoid biasing against the original annotations, we stuck to the original ones whenever there were ambiguities, and we double-checked each corrected annotation by additionally writing down reasons why we felt it was better. As mentioned in §4.1, we ultimately corrected 61 out of the 240 SQL annotations. §7.1 analyzes these corrections in greater detail.

6.3 Non-Expert Annotation

We split the 240 utterances into 8 units, each of which contains 30 utterances across 4–5 database domains and proved to take 1–2 hours to annotate with our interface (2–4 minutes per utterance). For the annotator behavior model $p(r \mid u, s, i)$ in equation (1), we assumed that every annotator would respond correctly with 0.7 probability and otherwise choose a response uniformly at random.

[11] We did not use these open-ended responses to generate SQL annotations. Future versions of the interface could explicitly design a mechanism to separately handle ambiguous, confusing, or unanswerable utterances.

Recruiting Non-Experts We recruited 11 university students who 1) were not pursuing/had not received a Computer Science degree and 2) had no prior experience with SQL to complete the annotation tasks. Each annotator could annotate any number of units (from 1 to 8) as they wished, but had to annotate them fully. For each unit we rewarded them with $15 as a base payment and a $5 ($10) bonus if their response agreed with our corrected gold standard $> 85\%$ (95%) of the time. We received 20 units of annotation in total, and hence each utterance was examined by 2.5 annotators on average. We asked each of them 1.84 questions about the utterance, on average. The databases that we presented contain only 8.71 records on average.

Participation Procedure We asked each annotator to: 1) sign a consent form to participate in the study, 2) watch a 12-minute video tutorial[12] that contains our annotation instructions and explains the basics of foreign and primary keys, 3) complete the annotation task,[13] and 4) fill out an exit survey about their major and their prior programming experience.

6.4 Learning an Annotator Behavior Model

After collecting the annotations, we used them to improve the annotator behavior model $p(r|u, s, i)$ of §6.3 by learning parameters α_a for each annotator a and β_d for each SPIDER domain d. For a given a and d, the chance that the annotator answers at random is no longer fixed at 0.3, but is modeled as $\sigma(\alpha_a + \beta_d + b)$, where σ is the logistic function and b is a bias term. Larger α_a and β_d predict higher error rates.

We maximize the incomplete-data log-likelihood, $L_u = \log \sum_s p(s, r_1, \ldots, r_T \mid u, i_1, \ldots, i_T)$, summed over all utterances u.[14] Since we explicitly model the annotator behavior, L_u is sensitive to the domain d of u and the annotator n_t who answered the question based on i_t. Just as in other adaptive crowdsourcing research (§8), we do not assume access to the gold value of s, so we impute it with $p_0(s)$. Our behavior model will predict a lower annotation error rate for those who tend to agree with other annotators and with Codex.

Overall, our annotators chose the actually correct option 72% of the time. To evaluate our unsupervised annotator model, we compared its predicted probability of a correct response on each utterance to the 0/1 indicator of whether the annotator did choose the correct option. Our model achieves an AUC ROC score of 0.68 and MSE error of 0.185. For comparison, a simple *supervised* model that always predicted the true overall probability of 72% would achieve an MSE error of 0.20.

We analyze our annotator behavior model in greater detail in Appendix H, and outline ways to improve it in Appendix A.

6.5 Results

After training the annotator behavior model as above, we recompute our posterior distribution for each utterance and evaluate the highest weighted candidate. On this dataset with 240 re-annotated utterances, the candidate ceiling is 88%; the interaction ceiling is 84%, which is calculated by interacting with a simulated perfect annotator. Our 1-best posterior candidate achieves 75% accuracy, significantly better than the 59% accuracy achieved by the 1-best candidate of the Codex prior.[15] This is comparable to the accuracy of the original SPIDER annotation performed by database experts (75%). A breakdown is shown in Appendix H as Table 3.

7 Analysis

7.1 Errors in Expert Annotations

APEL helped us identify annotation mistakes in the original SPIDER dataset (§6.2). We categorize and present them below to demonstrate where our framework is most helpful.

Ties for Extremals For the utterance "*Who is the youngest person?*", the SPIDER annotation is `SELECT NAME FROM PEOPLE ORDER BY AGE LIMIT 1`. As APEL discovers, in case of ties, humans prefer a SQL that will return *all* of the people who have the smallest age, rather than just the first one. 28 out of the 61 updated annotations fall into this category.

`INNER JOIN` vs. `LEFT JOIN` Suppose the utterance is "*List singer names and number of concerts for each singer.*" and the database contains a table of singers and a table with records (s, c) if

[12] URL: https://youtu.be/Z-kJW8upF8Y

[13] An example unit of the annotation task can be seen here: http://35.225.126.31:4200/v0104_4pm_8

[14] We omit the subscript u for r, s, and i.

[15] Recall from §4.2 that the prior (and hence also the posterior) involved prompting Codex with the noisy SPIDER annotations from the validation split, which did not quite match the style of our re-annotated evaluation dataset.

singer s performed in concert c. The SPIDER annotation only uses `INNER JOIN` and hence fails to return singers with count 0 (who have not performed in any concert). 8 of the updates fall into this category.

Interpreting Database Schema Properly Suppose each row contains information about an orchestra, including the year the orchestra was founded and its associated recording company. For an utterance "*Which recording company was founded the earliest?*", the correct annotation should be "Not enough information to tell". Neither SPIDER nor APEL currently supports this annotation option, though our human annotator reported this issue in their open-ended feedback. Notably, SPIDER provides a wrong annotation `SELECT company from TABLE WHERE YEAR = (SELECT MIN(YEAR) from TABLE)`, which looks plausibly correct but actually finds the recording company of the earliest-founded orchestra.

Handling all Allowed Values The annotated SQL should behave appropriately on all plausible cell values. For example, when we are asked about the maximum value in a column that allows `NULL` cells, we prefer a SQL that skips the `NULL` cells and returns the actual maximum value. As another example, if the utterance is "*How many countries have a republic government form?*", the clause `WHERE GOVERNMENT = "Republic"` will ignore any countries with the government form "*Federal Republic*"; the correct annotation should be `WHERE GOVERNMENT LIKE "%Republic%"`.

Nevertheless, it is difficult to handle arbitrary cell values allowed by the schema. For example, if a user asks "*how many dogs are there*", an ideal annotated SQL might need to account for cell values like "*Chihuahua*", "*Husky*", etc, which requires common-sense reasoning and is hard to specify in SQL. Therefore, we either need to make stronger assumptions about what cell values are allowed in a database, or introduce additional modules to handle common sense. During our re-annotation process in §6.2, we leniently considered the original SPIDER annotations to be correct as long as they could handle all values that appeared in SPIDER's sample database, even if the database schema would allow a larger set of possible cell values.

Remark Since most of the text-to-SQL models had low performance 3 years ago, Yu et al. (2018) favored short and plausible SQL annotations to make learning easier. These annotation conventions were shared between training and test sets to form a coherent structured prediction task (internal validity). Now that structured prediction is working well enough that the predictions could be used in real-world settings, we should turn to assuring that the SQL annotations actually have the desired effects (external validity). APEL can help in establishing the new gold standard (§6.2).

7.2 Errors in Non-Expert Responses

To inform future improvements for our framework, we categorize and present mistakes made by non-experts using APEL below.

Ambiguous Utterances Consider the utterance "*What are the names of properties that are either houses or apartments with more than 1 room?*" Should it be parsed as "*(house) or (apartment and room > 1)*", or "*(house or apartment) and room > 1*"? Another example: "*Count the number of friends Kyle has.*" What to do when there are two students named `Kyle`?

Database Constraints with Common Sense Database schemas sometimes omit common-sense constraints. For example, according to common sense, "`BIRTHDAY` + `AGE`" should always yield the current year, so sorting by `BIRTHDAY` ascendingly is equivalent to sorting by `AGE` descendingly. However, APEL is able to find a database where these two strategies return different outputs.[16] Such a database is unnatural and confuses humans.

Heavy Computation It is hard for humans to do arithmetic mentally, e.g., find the average of eight 9-digit values. To avoid demanding such computations, APEL should improve the annotator effort model $|i|$ beyond counting the number of records (Appendix B).

8 Related Work

Semantic Parsing Semantic parsers have improved significantly over the past decades (Zettlemoyer and Collins, 2007; Jia and Liang, 2016; Scholak et al., 2021a). Recent large pretrained models can perform the task without task-specific

[16] Even though APEL derives its databases from the original SPIDER sample database, that sample database contains records that do not conform to this unstated constraint.

architectures (Scholak et al., 2021b) or even in a zero/few-shot manner (Shin et al., 2021; Brown et al., 2020; Chen et al., 2021a). Rajkumar et al. (2022) evaluates Codex's text-to-SQL capability.

However, collecting semantic parsing datasets is still challenging since it requires experts. Wang et al. (2015) addresses this by synthetically generating logical forms, using templates to explain them in natural language, and asking crowdworkers to paraphrase them. Still, the paraphrases are usually restricted in linguistic diversity (Larson et al., 2020). Ideally we want to annotate programs based on naturally occurring utterances.

Programming by Example PBE has been applied to synthesize regular expressions (Gulwani, 2011), tensor manipulation (Shi et al., 2020), data analysis (Bavishi et al., 2019), and visualization (Wang et al., 2021) programs, etc. Future work can extend APEL to these applications as well, as we detail in §9.

Some other recent works such as Ye et al. (2020); Baik et al. (2020) also try to combine semantic parsing with PBE. However, both of them require the users to provide the input-output examples, which can be time-consuming to write. Pasupat and Liang (2016) asked non-programmers o-selection questions by synthesizing table inputs, but they did not optimize for question simplicity and focused on a simpler single-table setting. Ji et al. (2020) chooses questions to minimize the round of interactions, but did not optimize question simplicity or use a natural language prior.

Database Research Related to our work, Green et al. (2007) and Chu et al. (2017b) develop methods to prove semantic equivalence of SQLs, Wang et al. (2017a) synthesizes SQL from input-output examples, Chu et al. (2017a) searches for a database (counterexample) that makes two SQL return different values and Miao et al. (2019) minimizes the size of such a counterexample.

Active Learning For each utterance, we maintain a prior over the function (program) space, and learn the desired function by querying it on a sequence of carefully chosen inputs. Schulz et al. (2018); Ein-Dor et al. (2020); Karamcheti et al. (2021) apply active learning to real-valued functions, sequence classification, and visual question answering, respectively. In the medical domain, doctors use differential diagnosis (Henderson et al., 2012) to choose the most informative diagnostic tests to identify the underlying disease. More generally, one common goal of active learning is to choose input queries to reduce some expected loss (Settles, 2009). Our information gain criterion (4) specifically reduces the expected log-loss of the posterior, but this could be generalized (Appendix B).

Scalable Oversight As AI systems become more capable of generating candidate responses, an emerging line of research supervises AI systems by providing preferences over AI-generated candidates rather than providing human demonstrations (Stiennon et al., 2020; Wiegreffe et al., 2021; Askell et al., 2021; Liu et al., 2022; Ouyang et al., 2022). Therefore, to supervise AI to perform more complex tasks, it becomes increasingly important to determine human preferences over model outputs that are expensive to verify, such as full-book summaries or natural language descriptions of distributional properties (Amodei et al., 2016; Wu et al., 2021; Zhong et al., 2022). Our work presents a strategy to re-weight complex outputs (programs) from an AI system by asking simple informative questions of annotators who do not have to understand the outputs directly.

9 Discussion: Other Applications

Our framework can be generalized to other semantic parsing applications more broadly, where

- the SQL program s can be generalized to other types of executable semantic parses, such as tensor manipulation commands, visualization programs (Chen et al., 2021c), or dataflow graphs (Semantic Machines et al., 2020);
- the database schema c can be generalized to include any **c**ontext that affects the mapping of u to s, e.g., the conversational history preceding u, and the input type required by program s;
- the input database i can be generalized to any well-typed input if s is a function, or i can be the program state if s is a step in a procedural program;
- the database query result $o = s(i)$ can be generalized to the intended effect of u on i, which can include return values such as output tensors and images, or side effects such as file updates or robotic actions.

Applying APEL to a new type of semantic parsing application, rather than utterance-to-SQL, would require the following components:

A seed semantic parser that is likely to generate a short list of candidates that contain the correct program. This requirement is not hard to satisfy in many applications, given that large language models achieve often achieve high top-k accuracy on generating simple Python snippets (Chen et al., 2021a), JSON data (Poesia et al., 2022), Lispress (Shin et al., 2021) and SQL programs (Scholak et al., 2021b; Rajkumar et al., 2022) with only a few training examples and are likely to continue improving (Kaplan et al., 2020). For example, we achieved 95% top-32 accuracy on SPIDER without any task-specific engineering beyond few-shot prompting (e.g., specialized architectures (Wang et al., 2020), decoding constraints (Scholak et al., 2021b), etc).

An algorithm to find an simple informative program input i that satisfies c. Our method in §3 generates random databases by using an existing sample database as a reference and greedily drops rows to optimize the objective in equation (5). Future methods could potentially speed up the optimization process with a learned neural network (Chen et al., 2018) or a constraint solver (Chu et al., 2017a).

A graphical interface that enables the annotators to easily inspect the input i and choose the correct output o, where i, o can be generalized from database tables, strings, or numbers to calendar events (Andreas et al., 2020), voxel structures (Wang et al., 2017b), etc. Careful interface design (§6.1) can significantly reduce the effort required from the annotators.

In summary, APEL is a general framework for clarifying the semantics of natural language utterances. It elicits information from humans about how the semantic forms should behave when executed on particular inputs. In this paper we demonstrated the value of APEL on a text-to-SQL task. Some future work is given in Appendices A to B. It would also be desirable to improve our heuristics for the challenging problem of finding simple informative examples in the SQL domain. Finally, we look forward to future work that extends APEL to other semantic parsing applications.

Acknowledgements

The first author is funded by NSF-Simons Theorinet Grant (NSF Award #2031985). Our human interaction study was approved by UC Berkeley's Institutional Review Board, and our survey and interface did not collect any personal identifiable information. We thank members of the Berkeley NLP group for their feedback on an earlier draft.

References

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. 2016. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565*.

Jacob Andreas, John Bufe, David Burkett, Charles Chen, Josh Clausman, Jean Crawford, Kate Crim, Jordan DeLoach, Leah Dorner, Jason Eisner, Hao Fang, Alan Guo, David Hall, Kristin Hayes, Kellie Hill, Diana Ho, Wendy Iwaszuk, Smriti Jha, Dan Klein, Jayant Krishnamurthy, Theo Lanman, Percy Liang, Christopher H. Lin, Ilya Lintsbakh, Andy McGovern, Aleksandr Nisnevich, Adam Pauls, Dmitrij Petters, Brent Read, Dan Roth, Subhro Roy, Jesse Rusak, Beth Short, Div Slomin, Ben Snyder, Stephon Striplin, Yu Su, Zachary Tellman, Sam Thomson, Andrei Vorobev, Izabela Witoszko, Jason Wolfe, Abby Wray, Yuchen Zhang, and Alexander Zotov. 2020. Task-oriented dialogue as dataflow synthesis. *Transactions of the Association for Computational Linguistics*, 8:556–571.

Amanda Askell, Yuntao Bai, Anna Chen, Dawn Drain, Deep Ganguli, Tom Henighan, Andy Jones, Nicholas Joseph, Ben Mann, Nova DasSarma, et al. 2021. A general language assistant as a laboratory for alignment. *arXiv preprint arXiv:2112.00861*.

Yoram Bachrach, Thore Graepel, Thomas P. Minka, and Jo W. Guiver. 2012. How to grade a test without knowing the answers—a Bayesian graphical model for adaptive crowdsourcing and aptitude testing. In *ICML*.

Christopher Baik, Zhongjun Jin, Michael J. Cafarella, and H. V. Jagadish. 2020. Constructing expressive relational queries with dual-specification synthesis. In *CIDR*.

Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: Neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Anthony Chen, Pallavi Gudipati, Shayne Longpre, Xiao Ling, and Sameer Singh. 2021a. Evaluating entity disambiguation and the role of popularity in retrieval-based NLP. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4472–4485, Online. Association for Computational Linguistics.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Linyuan Gong, Alvin Cheung, and Dawn Song. 2021c. Plotcoder: Hierarchical decoding for synthesizing visualization code in programmatic context. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2169–2181.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Execution-guided neural program synthesis. In *International Conference on Learning Representations*.

Yuxin Chen, Shervin Javdani, Amin Karbasi, J. Andrew Bagnell, Siddhartha Srinivasa, and Andreas Krause. 2015. Submodular surrogates for value of information. In *Proceedings of AAAI*.

Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017a. Cosette: An automated prover for SQL. In *CIDR*.

Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017b. HoTTSQL: Proving query rewrites with univalent SQL semantics. *ACM SIGPLAN Notices*, 52(6):510–524.

A. P. Dempster, N. M. Laird, and D. B. Rubin. 1977. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, 39(1):1–21.

Liat Ein-Dor, Alon Halfon, Ariel Gera, Eyal Shnarch, Lena Dankin, Leshem Choshen, Marina Danilevsky, Ranit Aharonov, Yoav Katz, and Noam Slonim. 2020. Active Learning for BERT: An Empirical Study. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7949–7962, Online. Association for Computational Linguistics.

Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330.

Mark Henderson, Lawrence M. Tierney, and Gerald W. Smetana. 2012. *The Patient History: Evidence-Based Approach*. McGraw Hill Professional.

Ari Holtzman, Peter West, Vered Shwartz, Yejin Choi, and Luke Zettlemoyer. 2021. Surface form competition: Why the highest probability answer isn't always right. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7038–7051, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. 2020. Question selection for interactive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1143–1158.

Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12–22, Berlin, Germany. Association for Computational Linguistics.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Siddharth Karamcheti, Ranjay Krishna, Li Fei-Fei, and Christopher Manning. 2021. Mind your outliers! investigating the negative impact of outliers on active learning for visual question answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 7265–7281, Online. Association for Computational Linguistics.

Nate Kushman and Regina Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 826–836, Atlanta, Georgia. Association for Computational Linguistics.

Stefan Larson, Anthony Zheng, Anish Mahendran, Rishi Tekriwal, Adrian Cheung, Eric Guldan, Kevin Leach, and Jonathan K. Kummerfeld. 2020. Iterative feature mining for constraint-based data collection to increase data diversity and model robustness. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 8097–8106, Online. Association for Computational Linguistics.

Nada Lavrac and Saso Dzeroski. 1994. Inductive logic programming. In *WLP*, pages 146–160. Springer.

D. V. Lindley. 1956. On a measure of the information provided by an experiment. *Annals of Mathematical Statistics*, 27(4):986–1005.

Alisa Liu, Swabha Swayamdipta, Noah A. Smith, and Yejin Choi. 2022. Wanli: Worker and AI collaboration for natural language inference dataset creation. *arXiv preprint arXiv:2201.05955*.

David J. C. MacKay. 1992. Information-based objective functions for active data selection. *Neural Computation*, 4(4):590–604.

Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining wrong queries using small examples. In *Proceedings of the 2019 International Conference on Management of Data*, pages 503–520.

Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*.

Panupong Pasupat and Percy Liang. 2016. Inferring logical forms from denotations. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 23–32, Berlin, Germany. Association for Computational Linguistics.

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations*.

J. R. Quinlan. 1986. Induction of decision trees. *Machine Learning*, 1:81–106.

Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. Evaluating the text-to-SQL capabilities of large language models. *arXiv preprint arXiv:2204.00498*.

Torsten Scholak, Raymond Li, Dzmitry Bahdanau, Harm de Vries, and Chris Pal. 2021a. DuoRAT: Towards simpler text-to-SQL models. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1313–1321, Online. Association for Computational Linguistics.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021b. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Eric Schulz, Maarten Speekenbrink, and Andreas Krause. 2018. A tutorial on Gaussian process regression: Modelling, exploring, and exploiting functions. *Journal of Mathematical Psychology*, 85:1–16.

Semantic Machines, Jacob Andreas, John Bufe, David Burkett, Charles Chen, Josh Clausman, Jean Crawford, Kate Crim, Jordan DeLoach, Leah Dorner, Jason Eisner, Hao Fang, Alan Guo, David Hall, Kristin Hayes, Kellie Hill, Diana Ho, Wendy Iwaszuk, Smriti Jha, Dan Klein, Jayant Krishnamurthy, Theo Lanman, Percy Liang, Christopher H. Lin, Ilya Lintsbakh, Andy McGovern, Aleksandr Nisnevich, Adam Pauls, Dmitrij Petters, Brent Read, Dan Roth, Subhro Roy, Jesse Rusak, Beth Short, Div Slomin, Ben Snyder, Stephon Striplin, Yu Su, Zachary Tellman, Sam Thomson, Andrei Vorobev, Izabela Witoszko, Jason Wolfe, Abby Wray, Yuchen Zhang, and Alexander Zotov. 2020. Task-oriented dialogue as dataflow synthesis. *Transactions of the Association for Computational Linguistics*, 8:556–571.

Burr Settles. 2009. Active learning literature survey.

Kensen Shi, David Bieber, and Rishabh Singh. 2020. TF-Coder: Program synthesis for tensor manipulations. *arXiv preprint arXiv:2003.09040*.

Richard Shin, Christopher Lin, Sam Thomson, Charles Chen, Subhro Roy, Emmanouil Antonios Platanios, Adam Pauls, Dan Klein, Jason Eisner, and Benjamin Van Durme. 2021. Constrained language models yield few-shot semantic parsers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7699–7715, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Nisan Stiennon, Long Ouyang, Jeff Wu, Daniel M. Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul Christiano. 2020. Learning to summarize from human feedback. *arXiv preprint arXiv:2009.01325*.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *ACL*.

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017a. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 452–466.

Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J. Ko. 2021. Falx: Synthesis-powered visualization authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–15.

Sida I. Wang, Samuel Ginn, Percy Liang, and Christopher D. Manning. 2017b. Naturalizing a programming language via interactive learning. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 929–938, Vancouver, Canada. Association for Computational Linguistics.

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

Jacob Whitehill, Ting-fan Wu, Jacob Bergsma, Javier Movellan, and Paul Ruvolo. 2009. Whose vote should count more: Optimal integration of labels from labelers of unknown expertise. In *Advances in Neural Information Processing Systems*, volume 22. Curran Associates, Inc.

Sarah Wiegreffe, Jack Hessel, Swabha Swayamdipta, Mark Riedl, and Yejin Choi. 2021. Reframing human-AI collaboration for generating free-text explanations. *arXiv preprint arXiv:2112.08674*.

Jeff Wu, Long Ouyang, Daniel M. Ziegler, Nisan Stiennon, Ryan Lowe, Jan Leike, and Paul Christiano. 2021. Recursively summarizing books with human feedback. *arXiv preprint arXiv:2109.10862*.

Yan Yan, Romer Rosales, Glenn Fung, and Jennifer G. Dy. 2011. Active learning from crowds. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, page 1161–1168, Madison, WI, USA. Omnipress.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2020. Benchmarking multimodal regex synthesis with complex structures. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6081–6094, Online. Association for Computational Linguistics.

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 *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Luke Zettlemoyer and Michael Collins. 2007. Online learning of relaxed CCG grammars for parsing to logical form. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 678–687, Prague, Czech Republic. Association for Computational Linguistics.

Ruiqi Zhong, Charlie Snell, Dan Klein, and Jacob Steinhardt. 2022. Summarizing differences between text distributions with natural language. *arXiv preprint arXiv:2201.12323*.

Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic evaluation for text-to-SQL with distilled test suites. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 396–411, Online. Association for Computational Linguistics.

Supplementary Material

A Training on APEL Annotations

As APEL is an annotation method, its ultimate goal is to convert unlabeled utterances u into "supervised" training examples (u, s) for a semantic parser. In this appendix, we discuss how these training examples would be constructed and used.[17]

Training on MAP Annotations For each utterance u, APEL solicits indirect annotations and produces a posterior probability distribution p_T over programs. The maximum *a posteriori* (MAP) program is defined by $\hat{s} \stackrel{\text{def}}{=} \text{argmax}_s\, p_T(s)$.

Most simply, we could train a semantic parser on the inferred annotations $(u, \hat{s})$. Specifically, one option is to use these annotations to fine-tune the baseline parser p_0. However, this may not be practical for a very large pretrained model such as Codex, in which case the annotations could be used to train a dedicated semantic parser from scratch.

Downstream Evaluation In this paper, we evaluated APEL directly by asking whether its inferred annotations $(u, \hat{s})$ are accurate and hence useful for training. We measured this by semantic equivalence $[\![\hat{s}]\!] = [\![s^*]\!]$ (see §4.2) on a small test set of expert-labeled pairs (u, s^*) that were also annotated with APEL. However, one could instead actually train a parser on a large set of APEL-labeled utterances and then evaluate that parser on the same test set.

Beyond MAP Rather than focusing on only the MAP annotation of u, we could have evaluated APEL's entire distribution p_T using the log-loss, $-\log p_T(s^* \mid u)$, averaged over the test set of (u, s^*) pairs. This is a more sensitive evaluation, although harder to interpret than exact-match accuracy.

But do we care whether the distribution p_T is accurate beyond just its MAP annotation? Yes: one could train a parser p by maximizing its expected log-likelihood

$$\mathbb{E}_{s\sim p_T}[\log p(s \mid u)] \tag{7}$$

summed over the APEL-annotated utterances u. In effect, this treats p_T as giving a set of weighted "soft annotations" for u, not just the MAP annotation. It is equivalent to minimizing the Kullback-Leibler divergence $\text{KL}(p_T \mid\mid p)$ (summed over u).

Iterative Retraining Once an improved parser has been trained on the inferred annotations—either the MAP annotations or the soft annotations—it can be used as an improved prior p_0 in the update rule (1). This can inform APEL in the selection of future questions.

Furthermore, although APEL's past questions can no longer be changed, we can now reinterpret the annotators' responses to those questions, by using equation (1) to recompute an improved posterior p_T from the improved prior p_0. The improved soft annotations from these posteriors can be used to retrain the parser again. This procedure can be iterated to convergence to improve the parser.

This iterated procedure is a principled training method. If soft annotations are used for retraining via equation (7) at each iterations, then it is an instance of the Expectation-Maximization (EM) algorithm (Dempster et al., 1977). (If MAP labels are used, it is an instance of the hard or "Viterbi" approximation to EM.) EM is guaranteed to find a semantic parsing model p_0 that locally maximizes the conditional log-likelihood of all of the observed data—in our case, the annotators' responses r_t:

$$\begin{aligned}&\log p(r_1, \ldots r_T \mid u, i_1, \ldots i_T)\\ &= \log \sum_s p_0(s) \prod_{t=1}^{T} p(r_t \mid s(i_t))\end{aligned} \tag{8}$$

summed over all utterances u. Thus, it is simply a log-likelihood training method where the observed data are not direct annotations s, but indirect annotations r_t. It marginalizes over the latent program s.

Richer Model of Annotator Behavior The annotator model $p(r \mid s(i))$ can be jointly retrained with the semantic parser at each iteration of EM, to obtain a higher log-likelihood (8). Improving the annotator model in this way should result in more accurate distributions p_T at the next iteration of EM, and thus a better semantic parser.

Specifically, when we retrain the parser via equation (7), we can also retrain the annotator model to maximize the expected log-likelihood

$$\mathbb{E}_{s\sim p_T}[\sum_t \log p(r_t \mid s(i_t))] \tag{9}$$

[17] In the main paper, we used "example" to refer to examples of the form (i, o) used for programming by example, as mentioned in footnote 2. In this appendix, we use "example" to refer to examples of the form (u, s).

summed over the APEL-annotated utterances u. This EM procedure will maximize the incomplete-data log-likelihood as in §6.4.

We could fit a richer model of annotator behavior than the relatively simple one in §6.4. For example, we could estimate the error rates of individual annotators on different types of questions and program inputs. Intuitively, equation (9) means that we will tend to judge annotators as correct on examples where they agree with the consensus of p_0 and the other annotators.

Although equation (1) assumed that $p(r \mid u, s, i) = p(r \mid s(i))$, a rich model could drop this assumption.[18] For example, the model might allow that the annotator is more likely to make an error when u is difficult or i is complex. As an annotator's errors on a given utterance may be influenced by their previous errors or previous questions, a rich model could even take the form $p(r_t \mid u, s, i_1, r_1, \ldots, i_{t-1}, r_{t-1}, i_t)$.

Evaluating with Task Loss In our discussion of evaluation thus far, we have not given the semantic parser any partial credit. That is, given an utterance u, we have considered any answer other than the correct program s to be equally wrong.

However, more generally, it may be tolerable to find a program whose outputs are correct—or at least close to correct—for most inputs. Let $\text{loss}(\hat{o} \mid u, i, o^*)$ denote the task-specific loss of predicting output $\hat{o}$ on input i when the correct output is o^*. The loss of predicting program $\hat{s}$ when the correct program is s^* is then

$$\mathbb{E}_i[\text{loss}(\hat{s}(i) \mid u, i, s^*(i))] \tag{10}$$

where the expectation $\mathbb{E}_i$ is taken over some realistic distribution of inputs for utterance u. This loss function can be used in supervised evaluation.

Decoding with Task Loss Following standard Bayesian decision-theoretic methods, the semantic parser itself can aim to achieve low loss. No change to the training procedure is needed. Suppose we have trained a semantic parsing model $p(s \mid u)$ by EM as described above. We now need to extract decisions from this model. If the semantic parser is required to translate u into a single program $\hat{s}$ that will be used for all inputs, then the best program to choose is the program that minimizes the **Bayes risk**

$$R_p(\hat{s} \mid u) \stackrel{\text{def}}{=} \mathbb{E}_{s\sim p}[\mathbb{E}_i[\text{loss}(\hat{s}(i) \mid u, i, s(i))]] \tag{11}$$

This $\hat{s}$ is not necessarily the MAP program. Once $\hat{s}$ is predicted, the output on any given input i is $\hat{o} = \hat{s}(i)$.

In some applications, it may not be necessary to choose a single program. Then the best output $\hat{o}$ to return for input i is the output that minimizes the Bayes risk

$$R_p(\hat{o} \mid u, i) \stackrel{\text{def}}{=} \mathbb{E}_{s\sim p(\cdot|u)}[\text{loss}(\hat{o} \mid u, i, s(i))] \tag{12}$$

In other words, $\hat{o}$ is now predicted from i by a consensus of multiple programs.

B Extensions to APEL Annotation

The following natural extensions could be explored in future work.

Selecting Questions with Task Loss A model of task loss as in Appendix A can be used to improve the selection of the next question i_t. Instead of maximizing the expected reduction in entropy (equation (4)), we can maximize the expected reduction in the minimum Bayes risk (12).

When p is any distribution over programs s for utterance u, define the **minimum Bayes risk** by

$$\text{MBR}(p) \stackrel{\text{def}}{=} \min_{\hat{s}} R_p(\hat{s} \mid u) \tag{13}$$

At the start of round t, we can already achieve a Bayes risk of $\text{MBR}(p_{t-1})$. Once we have the annotator's response r_t to question i_t, we will be able to update p_{t-1} to p_t and achieve a Bayes risk of $\text{MBR}(p_t)$, where p_t depends on i_t and r_t via the update (2). Thus, the value of information from asking question i_t is

$$\text{MBR}(p_{t-1}) - \mathbb{E}_{r_t\sim p_{t-1}}[\text{MBR}(p_t)] \tag{14}$$

This is essentially the same as the information gain (4), but with task loss replacing log-loss. It is again guaranteed to be ≥ 0.

Richer Model of Annotator Effort Our objective (5) tries to present the annotator with "simple" input databases i_t, as measured by the number of records $|i_t|$. However, $|i_t|$ is only a crude measure of the time or effort that the annotator must expend to answer the multiple-choice question derived from i_t. For example, annotators typically

[18] This means using $p(r_t \mid u, s, i_t)$ in place of $p(r_t \mid s(i_t))$ in equations (2), (3), (8) and (9).

find it more difficult to perform arithmetic or to reason across multiple tables within i_t.

To predict the annotator's effort on the input database i_t, we could attempt to simulate the annotator's mental execution of the true program s on i_t.[19] As we do not know the true s, we would need to take the expectation of this effort over $s \sim p_{t-1}$. Mental operations such as adding 3-digit numbers or visually scanning the tables for a specific string could be then given appropriately high costs in the effort model.

In addition, perhaps a question that generates more multiple-choice options requires more effort. To capture this, the effort model could assign a cost not only to computing $s(i_t)$ but also to finding that answer in the list of options. More generally, the model could attempt to capture mental strategies that do not fully compute $s(i_t)$ but only do enough work to eliminate most of the options.

Finally, a question generally requires less human cognitive effort if it is related to a preceding question. Querying input database i with utterance u is easier for an annotator who has just queried the same i with a different u', or queried a different i' with the same u.

How would we train the parameters of the effort model? We observe how quickly the annotator answered each i_t with r_t. Taking this time as a proxy for effort, we can train the effort model to predict it from the true program s (which the annotator is presumed to know) and i_t. If we do not know s, we can impute it from the observed responses—that is, evaluate the effort model's log-likelihood in expectation under $s \sim p_T$, analogously to equations (7) and (9). In short, the effort model could be trained by EM, jointly with the other models.

Choosing Who Should Annotate What The selection objective (5) manages the tradeoff between annotator effort and information gain by imposing a hard constraint on the annotator effort per o-selection question. We could improve this crude selection objective—especially given good models of annotator behavior and annotator effort—by selecting a question i that achieves high "bang for the buck." This score is given by the *ratio* of value of information (equation (4) or (14)) to expected effort.

Both the numerator and denominator of this ratio depend on the specific annotator a who is answering i, if the models have annotator-specific parameters. They also depend on the utterance u. Thus, the score is actually a property of the triple (u, i, a).

At each step of APEL annotation, we would select a high-scoring triple—one in which we expect annotator a to usefully evaluate utterance u's on input database i with low effort. The annotator's response r then influences the scores of other triples involving u. We would stop annotation when no sufficiently high-scoring triple could be found.

In the same way, Bachrach et al. (2012) and Whitehill et al. (2009) model each individual annotator's capability and each question's difficulty, learning these parameters through agreement information. Yan et al. (2011) explore these ideas for active learning, similarly routing useful questions to competent annotators. Our experiment empirically finds that each annotator's ability to answer questions and the difficulty of each domain varies wildly (Figure 9); therefore, future work is likely to benefit from actively selecting who to annotate which utterances.

Non-Myopic Selection Policy The procedure described just above will switch freely among utterances and inputs, if it always selects the highest-scoring triple. However, recall that if a followup question is eventually to be asked at all, asking it immediately will incur less effort from the annotator. Thus, a good heuristic is to prioritize followup questions over non-followup questions, provided that they score highly enough that it is likely that they will eventually be asked.

More generally: Greedily choosing the highest-scoring question at each step is common in interactive protocols for information acquisition, such as adaptive quadrature or Bayesian optimization (Schulz et al., 2018). However, this greedy procedure is in general suboptimal. One could do better at selecting the next question by planning ahead to subsequent questions (Chen et al., 2015), though at a higher computational cost.

Again, we leave all these refinements to future work.

C Other Synthesis Constraints

This appendix gives further details about our method of database construction (§3). Overall, we follow the recipe of Zhong et al. (2020) to generate large informative databases that conform to a

[19] The same simulation could be also used to help model errors in the annotator's response r_t (Appendix A).

given schema c. We draw upon the existing sample database with this schema (provided by the SPIDER dataset in our experiments) to obtain plausible cell values. Following Zhong et al., we first synthesize cell values for all the "parent" columns (i.e., the columns that are being referenced by a child column in a foreign key relation), and then populate child columns with random elements from the parent columns.

Naturalness of i As shown in Figure 5a, unrestricted random cell values can confuse annotators who are unfamiliar with databases. Therefore, rather than synthesizing completely random values, we now always copy individual cell values from the sample databases (§4.1), optionally with minor perturbations such as ± 1 for integer values.

A database record might also be confusing even if its individual cell values are not. For example, the annotator can be confused by counterfactual information where the U.S. is in Asia as shown in Figure (b). Therefore, we prefer to initialize i^0 with the existing database. The annotator can also be confused by uncommon patterns where two people have the same name but different IDs; therefore, if the existing sample database has unique values in a column, we prefer to enforce that i also has unique values in that column.

Non-vacuous Execution Extremely small i frequently leads to undefined denotations. For example, since the maximum value for zero element is undefined, the correct denotation is `NULL`, which confuses annotators without a computer science background (Figure 5b). Therefore, we always add a small probability mass of `RETURN NULL` to the distribution p', which incentivizes our algorithm to produce i such that other SQL candidates will return non-`NULL` values.

Even if the returned value is well-defined, small i can lead to confusion if some operators are not needed to answer the question. For example, in Figure 5e, asking the maximum over one element might appear confusing, as we do not need the max operator to obtain a correct denotation. Therefore, we always add into p' a small probability mass of "neighbor queries" (Zhong et al., 2020) obtained by dropping aggregation operators and `WHERE` clauses from SQL candidates in p'. This incentivizes our algorithm to produce i such that the SQL candidates will meaningfully use their operators.

Managing Tradeoffs between two Criteria All the above tweaks make a tradeoff between the informative and the simplicity criteria in some way: we impose restrictions on i or modify p' to decrease the annotator effort while sacrificing information gain we can potentially achieve. How do we decide when to apply certain tweaks?

In our paper, we always add small probabilities of neighbor queries and `RETURN NULL` to p' and use cell values from the existing database. We then consider 3 types of tweaks that we apply if possible: 1) i_0 satisfies the uniqueness constraint, 2) i_0 is initialized with an existing database, and 3) $|i| \leq 15$ rather than 30. We apply these tweaks if they do not prevent us from succeeding. We define in total $2^3 = 8$ different "configurations" $0 \leq c < 8$, each of which specifies what subset of tweaks to apply to the algorithm described in §3. For example, $c = 6 = B110$ means we apply tweaks 1) and 2). We enumerate from $c = 7$ to 0 until the algorithm from §3 returns a program input with $\mathrm{IG}(i) \neq 0$. In other words, we start by applying all the tweaks and drop the tweaks gradually until we obtain a i with positive expected information gain.

D Fixing SPIDER Databases

We found several issues with the SPIDER databases and fixed them as follows:

- Some SPIDER databases do not conform to the foreign key constraint, i.e. some of the children columns contain values not in the parent columns they are referring to. We enforce the foreign key constraint by dropping the illegal records.
- We identify missing foreign key constraints under some domains and add them.
- The `voter_1` domain does not contain an appropriate foreign key design. Since fixing it would require an effort of re-annotating all 15 associated SQLs, we chose to exclude this domain from our evaluation.
- Some `Date` typed columns are string-valued and use English words to represent values, e.g. `nov1,2021`. As a result, `dec1,2021`, which is chronologically later, will be considered smaller alphabetically. We fix this by canonicalizing date representations into a `yyyy-mm-dd` format.

(a)

Name	Age	Section
asdg	102	^&(#@
qwerty	200	pqogen

Unnatural values

(b)

Country	Continent
U.S.	Asia
Canada	Africa

Counterfactual content

(c)

ID	Name	Age
1	Eren	26
2	Eren	23

Uncommon pattern

Utterance: *How old is the youngest person from section A?*

(d)

Name	Age	Section
Reiner	26	B

Vacuous answer → `NULL`

(e)

Name	Age	Section
Eren	26	A

Vacuous operator → `26`

Figure 5: Examples of unnatural databases (above) and vacuous execution (below), which motivates several tweaks in Appendix C. In (a) the individual cell values are unnatural. In (b) the records contradict world knowledge. In (c) the database contains two persons with the same name, which is atypical (but possible). In (d) the denotation of the utterance is undefined, since we cannot take the maximum over zero element. In (e) we do not need the max operator to obtain the correct denotation, since there is only one person in section A.

We accordingly update the suite of test cases from (Zhong et al., 2020) that we use to check whether two SQL forms are equivalent (see §4.2), so that they conform to the new database schema.

E Generating SQL Candidates

E.1 Prompting Codex

As sketched in §4.2, we obtain SQL program candidates through few-shot prompting, where the database schema is followed by 4 or 8 (with 50% probability) pairs of natural language utterances with their corresponding SQL queries from the SPIDER development set from the subset of utterance-SQL pairs associated with the same database schema. To select each in-context example, with probability 50% we choose a random example that has not been selected from the validation split, and with probability 50% we choose the most similar example that has not been selected based on TF-IDF similarity. Finally we append the target natural language utterance u to be annotated, and ask Codex to continue generating text after this prompt, which generates a candidate SQL program corresponding to u. An example prompt can be seen in Figure 6. We sampled 200 different prompts, which varied in their selected examples, and for each prompt we sampled 20 candidates from Codex with temperature=1.0 and top_p=0.95.

k	easy	medium	hard	extra	all
1	0.87	0.80	0.56	0.45	0.72
2	0.94	0.89	0.74	0.63	0.84
4	0.96	0.93	0.87	0.70	0.89
8	0.97	0.95	0.95	0.78	0.92
16	0.98	0.96	0.98	0.81	0.94
32	0.98	0.96	0.98	0.85	0.95

Table 1: The top-k accuracy for the SQL candidates generated by Codex on SPIDER (Yu et al., 2018), calculated on each split.

E.2 Top-k Accuracy

We report the top-k accuracy from the prior p_0 over SQL queries (§4.2) in Table 1, and graph the top-k accuracy curve in Figure 7.

F Computation

We did not compute the runtime in a controlled environment, so the statistics in this section are our best estimate.

Finding a single informative small database can take up to several minutes. The simulation evaluation on the evaluation split in §5 (524 utterances) takes around 240 CPU hours in total.

For the human evaluation in §6 (240 utterances), we must pre-compute the databases for each utterance, in order to support real-time interaction. Since we may ask the annotator up to 3 questions about the utterance, the choice of database i_t is conditioned on 0–2 previous questions and responses $(i_1, r_1, \ldots, i_{t-1}, r_{t-1})$. We pre-compute the database i_t for each of these possible histories that we may encounter.[20] This takes around 100 CPU hours in total (for the 240 utterances).

[20] Except that we set a timeout of 40 minutes per utterance. Of the 240 utterances, 7 utterances timed out before computing all of the databases in the response tree. (These primarily came from one domain where SPIDER's sample database was very large.) If during the interaction with an annotator, we needed a database i_t that we had not precomputed, we aborted the interaction early (we considered this as a failure to find an appropriate database, in the terms of §2.1). However, this rarely happened, since at each node of the response tree, we considered the most probable responses first.

```
CREATE TABLE Highschooler(
        ID int primary key,
        name text,
        grade int)
CREATE TABLE Friend(
        student_id int,
        friend_id int,
        primary key (student_id,friend_id),
        foreign key(student_id) references Highschooler(ID) ON DELETE CASCADE,
        foreign key (friend_id) references Highschooler(ID) ON DELETE CASCADE
)
[Other table schema omitted]

Write a query that answers "Count the number of high schoolers."
SELECT count(*) FROM Highschooler

Write a query that answers "What are the names of high schoolers who have 3 or more
friends?"
SELECT T2.name FROM Friend AS T1 JOIN Highschooler AS T2 ON T1.student_id  =  T2.id
GROUP BY T1.student_id HAVING count(*)  >=  3

[6 More examples omitted]

Write a query that answers "Find the average grade of all students who have some
friends."
_________[Models' Completion] ______
```

Figure 6: An example prompt we use for the Codex API. We obtain SQL program candidates through 4/8-shot prompting, where the database schema (orange) is followed by 4/8 pairs of natural language utterance and their corresponding SQL queries from the SPIDER development set, randomly sampled from the subset of queries associated with the same database schema. Finally we concatenate the target natural language utterance u to be annotated, and ask Codex to complete the prompt, which results in a candidate SQL program corresponding to u.

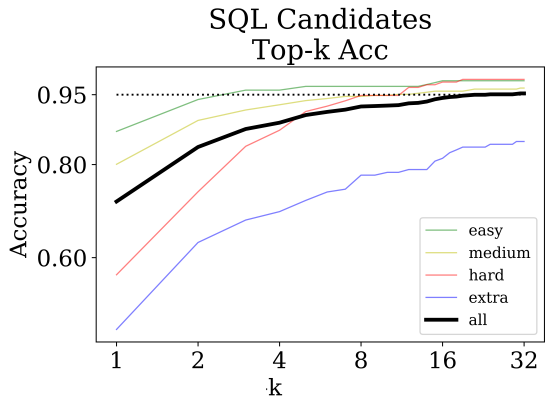

Figure 7: The top-k accuracy for the candidate SQL programs generated by Codex, after filtering and merging candidates. On each difficulty split we plot the curve of top-k accuracy (y-axis) and k (x-axis, log-scaled). The numbers can be seen in Appendix Table 1.

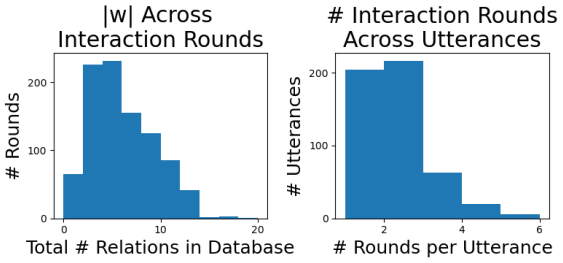

Figure 8: The interaction statistics using the SPIDER annotation to simulate an ideal annotator. **Left**: the distribution of the size of the database c across each round of interaction. **Right**: the distribution of the number of rounds across difference utterance.

G Simulated Interaction Statistics

The breakdown statistics of candidate and interaction ceiling (see §4.3) can be seen in Table 2, and the distribution database sizes and number of rounds of interaction can be seen in Figure 8.

H Human Annotation

We provide breakdown statistics on the annotation accuracy of different sources based on difficulty in

	easy	med	hard	extra	all
Candidate	0.99	0.97	0.98	0.88	0.96
OurDB	0.93	0.95	0.97	0.75	0.91
OrigDB	0.98	0.90	0.83	0.66	0.86

Table 2: The accuracy ceilings on each difficulty split. "med" stands for medium difficulty. Candidate means the candidate ceiling. OurDB means the accuracy ceiling achieved by querying an oracle annotator with at most 3 small informative databases we synthesize. OrigDB means the accuracy ceiling by querying with the databases released by the SPIDER dataset.

Table 3.

More analysis on the annotator behavior model can be found in Figure 9.

I Interface

See Figure 10 for a detailed screenshot of our interface. We implemented the front-end of our interface with Angular and the back-end was built with flask and Redis. Users are presented with a sequence of 40 distinct questions, and each question may have multiple rounds. For each round, the user is given a 4 minute time-limit before the interface automatically transitions to the next question. Before being asked questions on a new database, the user is presented with a page displaying all the tables in the database alongside descriptions we wrote for each table (see Figure Figure 11 for an example screenshot). When answering questions, the user is given a link back to this page for reference.

The user can either select one of the multiple choice questions presented or select "No Answer is Correct", and depending on their selection the user is presented with a differing set of followup questions. Regardless of their selection, we always ask the user two optional followups: "if you think the question is ambiguous, tell us why." and "if the question looks confusing, tell us why." In addition to these optional questions, we sometimes ask required followup questions. Specifically, if the user is on their final round and selects an answer which does not agree with the SPIDER annotation, we ask them why they did not select the correct answer according to spider. Or if the user selects "No Answer is Correct", we ask "What is the answer you have in mind and why?" We use the users' answers to these followups to collect information on the users' reasoning in answering questions and to determine issues with the SPIDER dataset.

We implemented a number of features in our interface to minimize the annotator effort. One of the largest challenges in this task is answering questions across several foreign keys. We implement two distinct mechanisms to make this easier for users. Firstly we highlight all table values or foreign keys matching the value the mouse is currently hovering over. Secondly, we give the user the option to merge all foreign keys into a single table by pressing a "merge" button. We allow the users to choose when to merge because there is a trade-off; while merged mode can make reasoning about foreign keys easier, it also can significantly increase the width of the tables visible to the user.

Sometimes there are tables presented to the user that are not necessary for answering the question, so we give users the option to collapse tables to simplify their display.

J Video Transcript

Page 1 In this task, you will be asked to answer questions from several tables.

Page 2 Here is the overall idea. You will be given a question on the top of the page, several tables on the left of the page, and you need to choose one of the options on the right, that corresponds to the correct answer. In this question, you are asked to "Show name, country, age for all singers ordered by age from the oldest to the youngest." Therefore, we expect the correct option to list the information about Joe Sharp first, since he is older. We look at the options and B is correct. Notice that A is wrong because it does not list the information for all singers, and C is wrong because it lists the singers from the youngest to the oldest.

After you submit the answer, our system will ask you whether there is anything that appears ambiguous or confusing. We don't need it for this question now.

Page 3 Let's go through some more examples.

Page 4 In this question you are asked "How many singers do we have?" This is a tricky question. First notice that the tables have changed from before, so you need to re-read the table. Secondly, there are actually two singers, but they have the same name. You should consider them to be two different persons with the same name but different SSN, and hence choose B.

There is a time limit shown at the top of the page, and after 4 minutes the system will move on to the next question.

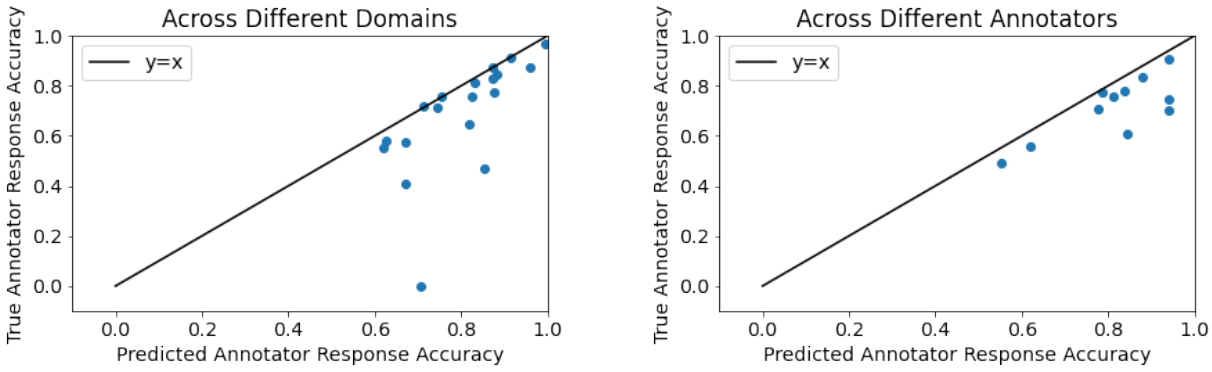

Figure 9: For each domain (left) and annotator (right), we use our annotator behavior model to predict the annotation accuracy (x-value) without using the gold annotation and compare it with the true annotation accuracy (y-value) using the gold annotation. The outlier at the bottom of the left figure corresponds to a domain that only contains two utterances.

	easy	medium	hard	extra	all
Candidate Ceiling	0.91	0.91	0.92	0.69	0.88
SPIDER	0.93	0.78	0.63	0.50	0.75
Codex	0.78	0.65	0.43	0.36	0.59
APEL ^{r}	0.75	0.71	0.65	0.49	0.67
APEL ^{m}	0.83	0.80	0.73	0.47	0.75

Table 3: 1-best accuracy of various SQL prediction methods, broken down by the difficulty level of the utterance (as categorized by SPIDER). Codex returns the most probable SQL according to p_0. APEL r does the same, after eliminating SQLs that are inconsistent with the responses of a single randomly chosen annotator. APEL ^{m} is our full method, which returns the SQL with the highest posterior probability after we fit our model of annotator error.

Page 5 Takeaways:

- Names are different from IDs. Two different people can have the same name.
- There is a time limit of 4 minutes for each question.

Page 6 In this question you are asked to find the song names of the singers above the average age. The average age is the mean of these 4 numbers, which is 34.5. The singers John and Rose have age above 34.5, so we can find their songs, which are sun and gentle man, which is D. Use a calculator if you need to!

Also, notice that there are other tables, but they are not relevant to the question. Feel free to ignore them. You can also choose to collapse them if that makes it easier, and you can click the button again to view it.

Page 7 Takeaways:

- Use a calculator if you need to.
- Not every table is needed.

Page 8 Here's the same question and the same table. Let's say somehow Sun and Gentleman is not one of the options, and then you should report that no answer is correct. Then we will ask you why you think no answer is correct. For example, you can write "gentleman and sun is correct. the average age is 34.5, John and Rose are above this age and have song gentleman and Sun".

The system asks us why we didn't choose A, we can answer "sun is by Rose, who is older than 34.5". Please tell us enough information so that we can know why your choice is correct - for example if you just say "sun is also a correct answer", it only describes the difference between the two options rather than explaining why it is correct. Giving us more information can help you win more bonus.

Page 9 Takeaways:

- Choose no option is correct and tell us why when you think no options are correct
- Tell us why you didn't choose an answer when we ask you to do so.

Page 10 The question is "What are the full names of all players, sorted by birth date?" First, notice

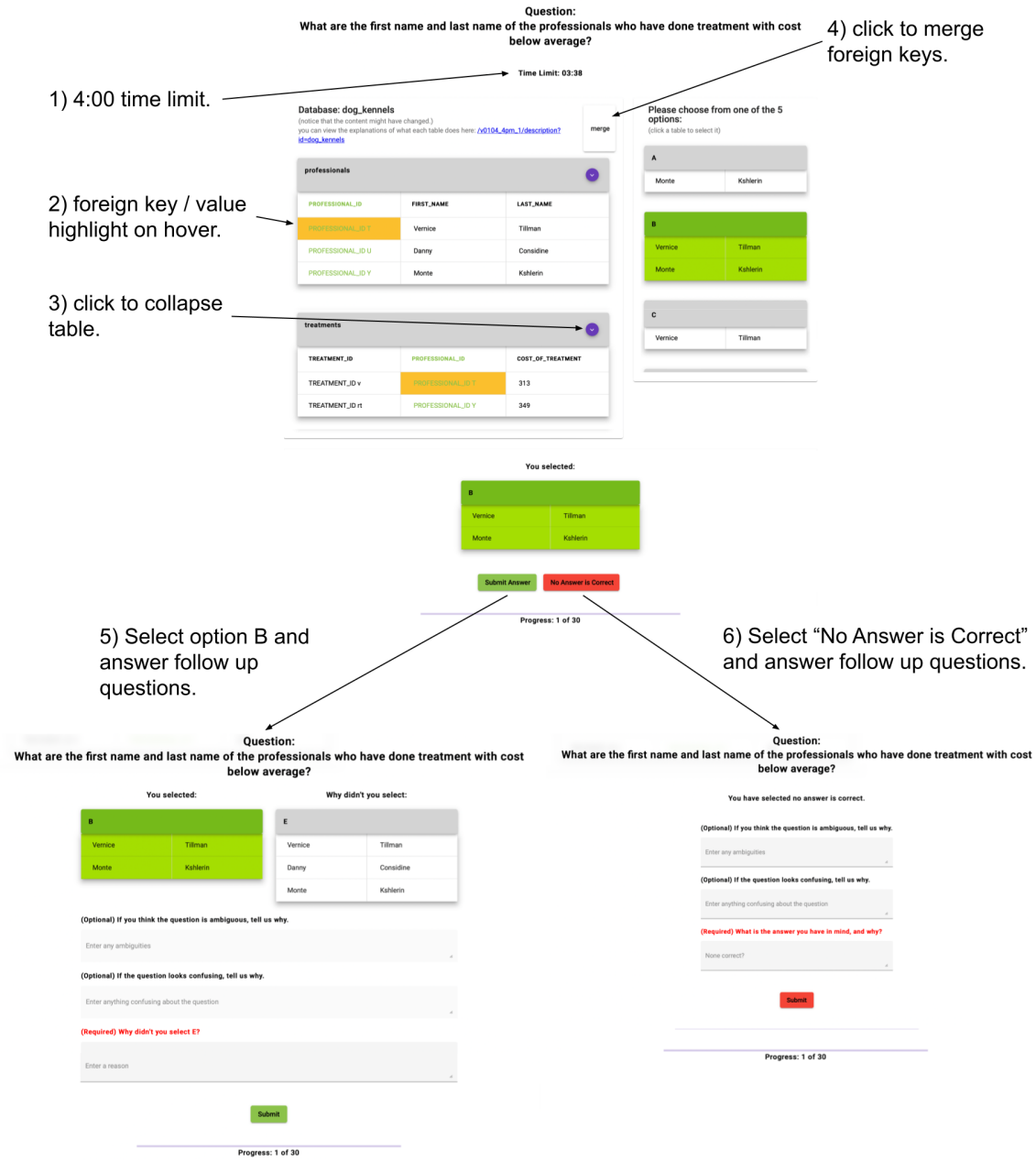

Figure 10: A detailed screenshot of our interface, and the logical flow of follow up questions.

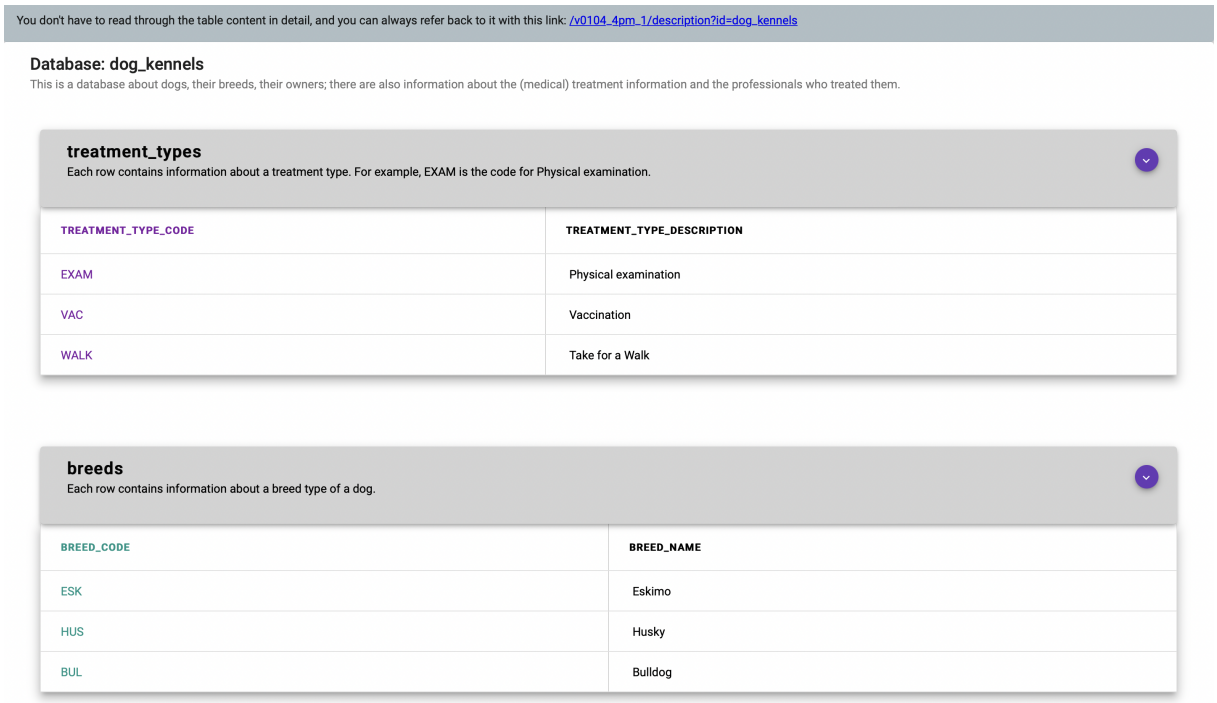

Figure 11: An example database description page presented to users before they start answering questions for that database.

that there are a lot of answers in this case, and you need to scroll down to read through all of them. Secondly, there are a lot of ambiguities: for example, the question didn't mention whether we should sort from youngest to oldest, or the reverse; secondly, the question does not mention whether the first and last name should be in the same column. For these reasons, A, B are both correct. C, D are wrong because the question does not ask for birthday information; F is wrong because it only lists one player and G is wrong for including birthday information. Then we can write in the response: "ABE are all correct; not sure if we should sort them from the oldest to youngest or reverse; also not sure whether to put the first and last name into the same column." But still, make your best guess, let's say, A.

Then we click submit, and the system asks us why we didn't choose C. We explain that "the question does not ask us for the birthday and it contains redundant information".

Page 11 Takeaways:

- There can be a lot of options. Make sure to read through every of them
- When the question is ambiguous and multiple answers are plausible, tell us why it is ambiguous and what are the plausible answers. But still, first make your best guess and submit.

Page 12 The question is "Give the names of countries that are in Europe and have a population equal to 80000." In this fictitious table, Brazil is in Europe and has a population of 80,000. Therefore, the correct answer is A, even though we know that Brazil is in fact in South America. However, it still cannot stop us from answering the question based on the table. Finally, there are many more countries in the world, beyond these three countries in the table, but we should pretend that there are only three countries in the world here.

Page 13 Takeaways:

- Try accepting the information from this table as much as possible and focus on the part useful for answering the question.
- If something is not present in the tables, pretend that it does not exist.

Page 14 Here are some more difficult tables.This is a database that contains information about battles and death. The overall description of the databases can be seen at the top of the page, which says: This database contains information about battles, death events, and ships. And then each table has its own

description as well. For example, in the ship table, each row contains information about a ship, the 4th row means the ship D was lost in battle with ID 4, and you can look up information about battle 4 in the battle table. To make it convenient for you, whenever you move your cursor to a value, all the same values will be highlighted. Here we notice that according to the 5th row, Ship E was also lost in battle 4.

To view multiple tables at the same time, you can choose to zoom out, like this. Then you can zoom back in, like this. You can typically find this option in the Help panel of your browser. Again, if you think some tables are irrelevant, just collapse them like this.

You don't have to study the tables in detail, since they will probably change for the next question.

Page 15 Takeaways:

- You don't have to study the table content in great detail, since they will be changing.
- Zoom-in/out if you need to. You can find them in the helper panel of your browser.

Page 16 This question is "Show names, results and bulgarian commanders of the battles with no ships lost in the 'English Channel".

The question asks for certain battles namely, those that did not lose ships in the English Channel [pause]. Let's start by finding the battles that did lose ships in the English channel [pause]. Only Battle 5 did; it lost ship C there. So the other battles, Battles 0 and 7, lost no ships there. In fact, Battle 0 lost no ships at all, which is why it doesn't show up in the second table. We find the names of Battle 0 and 7, along with their other information. Therefore, the answer is E. One very common mistake people make is that they ignored the word "no", and they chose the battles that lost the ship. Be careful and pay close attention to every word!

Notice that there was originally the death table. We removed it from the display to make it easier for you.

The phrase 'Bulgarian commander' might send you looking for a table that tells you each commander's nationality. But actually, Bulgarian_commander is a column in the battles table. Presumably this table lists battles that Bulgaria fought. Each battle had two sides, and this column is naming the commander for the Bulgarian side. You don't have to fully understand how the tables are set up, but you should figure out enough to answer the question.

Just to repeat, to make it easier for you to process this information, whenever your cursor moves to an ID or a piece of text, its counterpart in other tables will light up; whenever you click on a text, the counterpart in the answer will also be highlighted.

You can also choose to merge the tables. After you merge the table, there will still be two tables. Each of the rows in the battle table will still contain information about a battle, and each of the rows in the ship table will still contain information about a ship. However, the battle information where the ship is lost is merged into the ship table. Notice that battle 0 will not appear in the ship table, because no ship is lost in the battle, so be careful when you try to interpret the merged table. Click unmerge to recover to the original view.

Finally, if you forgot what each table means, you can always view them here.

Page 17 Takeaways:

- Pay close attention to how the question is being asked. They might lead to different options. Many mistakes people make are because they did not read the questions carefully.
- Sometimes we choose not to show you certain tables and columns if we know for sure they are not needed.
- Use the highlight functionality if that helps you to reason across tables.
- Use the merge functionality if you need to. Each table will contain information about the same object/entity, but the information about its related objects will be pooled in.

Page 18 The question is "List the name and date of the battle that has lost the ship named 'Lettice' and the ship named 'HMS Atalanta'." Since there is no ship named "HMS atlanta", there is no battle that lost both of these ships. So you should choose A, "no result found".

Page 19 Takeaways: Choose no_result_found if no answer satisfies the question.

Page 20 To summarize, here are a couple of things you need to remember to answer the questions correctly:

- Pay close attention to how the question is asked; most mistakes are made because of not reading the question carefully.
- Accept the information in the table even if they are changing and might be different from the knowledge you have for the real world
- IDs are different from names
- Some questions might have a lot of options to choose from and you need to read through all of them.

Page 21 To make it easier for you to answer the questions:

- Use the highlight and merge operations when you need to
- Use a calculator if you need to
- Zoom out to fit the tables into the screen and prevent scrolling.
- Not all table or column is needed to answer the questions

Page 22 For freeform response:

- Reporting ambiguities or tell us why the question is confusing only if you need to
- Explaining why you did not choose another option when we ask you. Giving us more information can you help you win more bonus.