Grounded Adaptation for Zero-shot Executable Semantic Parsing

Victor Zhong
University of Washington
Seattle, WA
vzhong@cs.washington.edu

Mike Lewis
Facebook AI Research
Seattle, WA
mikelewis@fb.com

Sida I. Wang
Facebook AI Research
Seattle, WA
sidawang@fb.com

Luke Zettlemoyer
University of Washington
Facebook AI Research
Seattle, WA
lsz@cs.washington.edu

Abstract

We propose Grounded Adaptation for Zero-shot Executable Semantic Parsing (GAZP) to adapt an existing semantic parser to new environments (e.g. new database schemas). GAZP combines a forward semantic parser with a backward utterance generator to synthesize data (e.g. utterances and SQL queries) in the new environment, then selects cycle-consistent examples to adapt the parser. Unlike data-augmentation, which typically synthesizes unverified examples in the training environment, GAZP synthesizes examples in the new environment whose input-output consistency are verified. On the Spider, Sparc, and CoSQL zero-shot semantic parsing tasks, GAZP improves logical form and execution accuracy of the baseline parser. Our analyses show that GAZP outperforms data-augmentation in the training environment, performance increases with the amount of GAZP-synthesized data, and cycle-consistency is central to successful adaptation.

1 Introduction

Semantic parsers (Zelle and Mooney, 1996; Zettlemoyer and Collins, 2005; Liang et al., 2011) build executable meaning representations for a range of tasks such as question-answering (Yih et al., 2014), robotic control (Matuszek et al., 2013), and intelligent tutoring systems (Graesser et al., 2005). However, they are usually engineered for each application environment. For example, a language-to-SQL parser trained on an university management database struggles when deployed to a sales database. How do we adapt a semantic parser to new environments where no training data exists?

We propose **G**rounded **A**daptation for **Z**ero-shot Executable Semantic **P**arsing, which adapts existing semantic parsers to new environments by synthesizing new, cycle-consistent data. In the previous example, GAZP synthesizes high-quality sales questions and SQL queries using the new sales database, then adapts the parser using the synthesized data. This procedure is shown in Figure 1. GAZP is complementary to prior modeling work in that it can be applied to any model architecture, in any domain where one can enforce cycle-consistency by evaluating equivalence between logical forms. Compared to data-augmentation, which typically synthesizes unverified data in the training environment, GAZP instead synthesizes consistency-verified data in the new environment.

GAZP synthesizes data for consistency-verified adaptation using a forward semantic parser and a backward utterance generator. Given a new environment (e.g. new database), we first sample logical forms with respect to a grammar (e.g. SQL grammar conditioned on new database schema). Next, we generate utterances corresponding to these logical forms using the generator. Then, we parse the generated utterances using the parser, keeping those whose parses are equivalent to the original sampled logical form (more in Section 2.4). Finally, we adapt the parser to the new environment by training on the combination of the original data and the synthesized cycle-consistent data.

We evaluate GAZP on the Spider, Sparc, and CoSQL (Yu et al., 2018b, 2019a,b) language-to-SQL zero-shot semantic parsing tasks which test on unseen databases. GAZP improves logical form and execution accuracy of the baseline parser on all tasks, successfully adapting the existing parser to new environments. In further analyses, we show that GAZP outperforms data augmentation in the training environment. Moreover, adaptation performance increases with the amount of GAZP-synthesized data. Finally, we show that cycle-consistency is critical to synthesizing high-quality examples in the new environment, which in turn allows for successful adaptation and performance

Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, pages 6869–6882,
November 16–20, 2020. ©2020 Association for Computational Linguistics

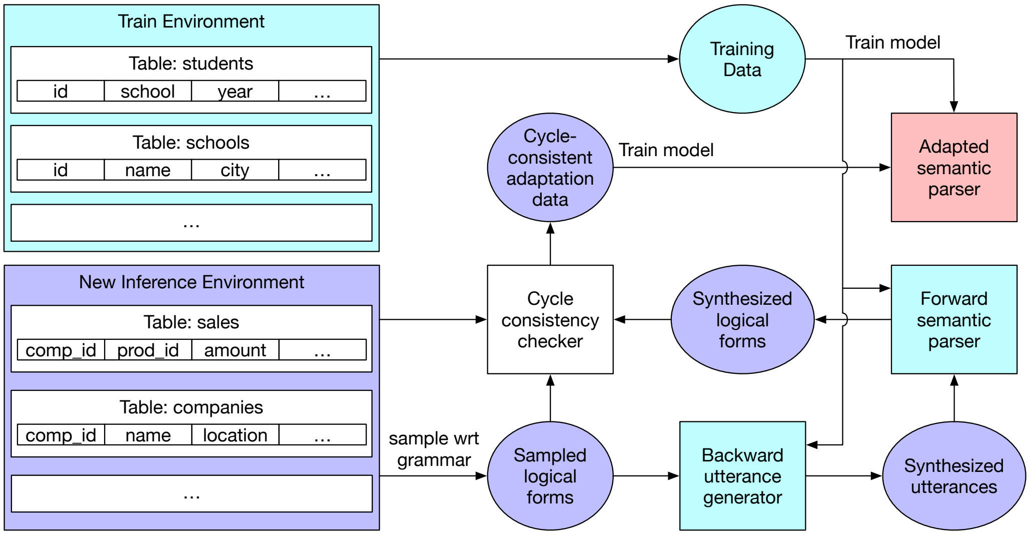


Figure 1: Grounded Adaptation for Zero-shot Executable Semantic Parsing. GAZP adapts a parser to new inference environments. Data and models for training and inference environments are respectively shown in blue and purple. Output is shown in red. First, we train a parser and a utterance generator using training data. We then sample logical forms in the inference environment and generate corresponding utterances. We parse the generated utterances and check for cycle-consistency between the parse and the sampled logical form (see Section 2.4). Consistent pairs of utterance and logical form are used to adapt the parser to the inference environment.

improvement.[1]

2 Grounded Adaptation for Zero-shot Executable Semantic Parsing

Semantic parsing involves producing a **logical form** q that corresponds to an input **utterance** u, such that executing q in the **environment** e produces the desired **denotation** $\textsf{Exe}(q, e)$. In the context of language-to-SQL parsing, q and e correspond to SQL queries and databases.

We propose GAZP for zero-shot semantic parsing, where inference environments have not been observed during training (e.g. producing SQL queries in new databases). GAZP consists of a **forward semantic parser** $F(u, e) \rightarrow q$, which produces a logical form q given an utterance u in environment e, and a **backward utterance generator** $G(q, e) \rightarrow u$. The models F and G condition on the environment by reading an **environment description** w, which consists of a set of **documents** d. In the context of SQL parsing, the description is the database schema, which consists of a set of table schemas (i.e. documents).

We assume that the logical form consists of three types of tokens: **syntax candidates** c_s from a fixed syntax vocabulary (e.g. SQL syntax), **environment candidates** c_e from the environment description (e.g. table names from database schema), and **utterance candidates** c_u from the utterance (e.g. values in SQL query). Finally, c_e tokens have corresponding spans in the description d. For example, a SQL query q consists of columns c_e that directly map to related column schema (e.g. table, name, type) in the database schema w.

In GAZP , we first train the forward semantic parser F and a backward utterance generator G in the training environment e. Given a new inference environment e', we sample logical forms q from e' using a grammar. For each q, we generate a corresponding utterance $u' = G(q, e')$. We then parse the generated utterance into a logical form $q' = F(u', e')$. We combine cycle-consistent examples from the new environment, for which $q'$is equivalent to q, with the original labeled data to retrain and adapt the parser. Figure 1 illustrates the components of GAZP. We now detail the sampling procedure, forward parser, backward generator, and cycle-consistency.

2.1 Query sampling

To synthesize data for adaptation, we first sample logical forms q with respect to a grammar. We begin by building an empirical distribution over q using the training data. For language-to-SQL parsing, we preprocess queries similar to Zhang et al. (2019) and further replace mentions of columns and values with typed slots to form **coarse**

[1] We will open-source our code.

Algorithm 1 Query sampling procedure.

```
$d \leftarrow$ UniformSample($AllDBs$)
$Z \leftarrow \emptyset$
for $z \in CoarseTemplates$ do
    if $d$.CanFill($z$) then $Z$.Add($z$) end if
end for
$z' \leftarrow$ Sample($P_Z$)
$d' \leftarrow d$.randAssignColsToSlots()
for $s \in z'$.colSlots() do
    $c \leftarrow d'$.getCol($s$)
    $z'$.replSlotWithCol($s, c$)
end for
for $s \in z'$.valSlots() do
    $c \leftarrow d'$.getCol($s$)
    $v \leftarrow c$.uniformSampleVals()
    $z'$.replSlotWithVal($s, v$)
end for
                                        ▷ Return $z'$
```

templates Z. For example, the query `SELECT T1.id, T2.name FROM Students AS T1 JOIN Schools AS T2 WHERE T1.school = T2.id AND T2.name = 'Highland Secondary'`, after processing, becomes `SELECT key1, text1 WHERE text2 = val`. Note that we remove `JOINS` which are later filled back deterministically after sampling the columns. Next, we build an empirical distribution P_Z over these coarse templates by counting occurrences in the training data. The sampling procedure is shown in Algorithm 1 for the language-to-SQL example. Invalid queries and those that execute to the empty set are discarded.

Given some coarse template z = `SELECT key1, text1 WHERE text2 = val`, the function d.CanFill(z) returns whether the database d contains sufficient numbers of columns. In this case, at the minimum, d should have a key column and two text columns. The function d.randAssignColsToSlots() returns a database copy d' such that each of its columns is mapped to some identifier `text1`, `key1` etc.

Appendix A.1 quantifies query coverage of the sampling procedure on the Spider task, and shows how to extend Algorithm 1 to multi-turn queries.

2.2 Forward semantic parser

The forward semantic parser F produces a logical form $q = F(u, e)$ for an utterance u in the environment e. We begin by cross-encoding u with the environment description w to model coreferences. Since w may be very long (e.g. entire database schema), we instead cross-encode u with each document d_i in the description (e.g. each table schema) similar to Zhang et al. (2019). We then combine each environment candidate $c_{e,i}$ across documents (e.g. table columns) using RNNs, such that the final representations capture dependencies between c_e from different documents. To produce the logical form q, we first generate a logical form template $\hat{q}$ whose utterance candidates c_u (e.g. SQL values) are replaced by slots. We generate $\hat{q}$ with a pointer-decoder that selects among syntax candidates c_s (e.g. SQL keywords) and environment candidate c_e (e.g. table columns). Then, we fill in slots in $\hat{q}$ with a separate decoder that selects among c_u in the utterance to form q. Note that logical form template $\hat{q}$ is distinct from coarse templates z described in sampling (Section 2.1). Figure 2 describes the forward semantic parser.

Let u denote words in the utterance, and d_i denote words in the ith document in the environment description. Let $[a; b]$ denote the concatenation of a and b. First, we cross-encode the utterance and the document using BERT (Devlin et al., 2019), which has led to improvements on a number of NLP tasks.

$$\overrightarrow{B}_i = \text{BERT}_{\rightarrow}([u; d_i]) \tag{1}$$

Next, we extract environment candidates in document i using self-attention. Let s, e denote the start and end positions of the jth environment candidate in the ith document. We compute an intermediate representation x_{ij} for each environment candidate:

$$a = \text{softmax}(W[\overrightarrow{B}_{is}; \ldots \overrightarrow{B}_{ie}] + b) \tag{2}$$

$$x_{ij} = \sum_{k=s}^{e} a_k \overrightarrow{B}_{ik} \tag{3}$$

For ease of exposition, we abbreviate the above self-attention function as $x_{ij} = \text{selfattn}(\overrightarrow{B}_i[s : e])$ Because x_{ij} do not model dependencies between different documents, we further process x with bidirectional LSTMs (Hochreiter and Schmidhuber, 1997). We use one LSTM followed by self-attention to summarize each ith document:

$$\overrightarrow{h}_{\text{enc},i} = \text{selfattn}(\text{BiLSTM}([x_{i1}; x_{i2}; \ldots])) \tag{4}$$

We use another LSTM to build representations for each environment candidate $c_{e,i}$

$$c_e = \text{BiLSTM}([x_{11}; x_{12}; \ldots x_{21}; x_{22} \ldots]) \tag{5}$$

We do not share weights between different LSTMs and between different self-attentions.

Next, we use a pointer-decoder (Vinyals et al., 2015) to produce the output logical form template

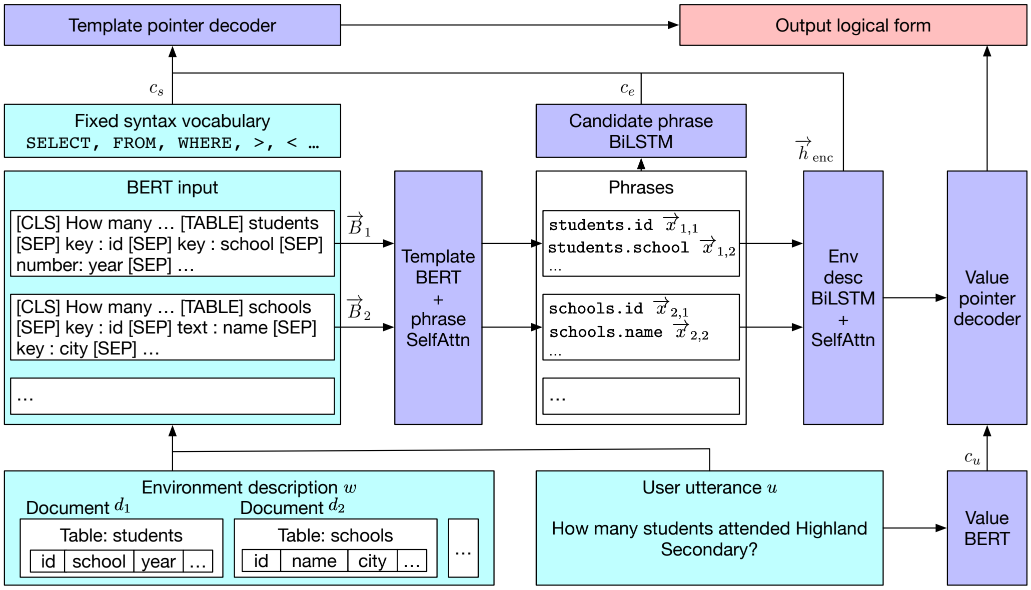

Template pointer decoder
Output logical form
c_s
c_e
Fixed syntax vocabulary
SELECT, FROM, WHERE, >, < …
Candidate phrase BiLSTM
$\overrightarrow{h}_{\text{enc}}$
BERT input
[CLS] How many … [TABLE] students [SEP] key : id [SEP] key : school [SEP] number: year [SEP] …
[CLS] How many … [TABLE] schools [SEP] key : id [SEP] text : name [SEP] key : city [SEP] …
…
$\overrightarrow{B}_1$
$\overrightarrow{B}_2$
Template BERT + phrase SelfAttn
Phrases
students.id $\overrightarrow{x}_{1,1}$
students.school $\overrightarrow{x}_{1,2}$
…
schools.id $\overrightarrow{x}_{2,1}$
schools.name $\overrightarrow{x}_{2,2}$
…
…
Env desc BiLSTM + SelfAttn
Value pointer decoder
c_u
Environment description w
Document d_1
Document d_2
Table: students
id school year …
Table: schools
id name city …
…
User utterance u
How many students attended Highland Secondary?
Value BERT

Figure 2: Forward semantic parser. Model components are shown in purple, inputs in blue, and outputs in red. First, we cross-encode each environment description text and the utterance using BERT. We then extract document-level phrase representations for candidate phrases in each text, which we subsequently encode using LSTMs to form input and environment-level candidate phrase representations. A pointer-decoder attends over the input and selects among candidates to produce the output logical form.

$\hat{q}$ by selecting among a set of candidates that corresponds to the union of environment candidates c_e and syntax candidates c_s. Here, we represent a syntax token using its BERT word embedding. The representation for all candidate representations $\overrightarrow{c}$ is then obtained as

$$\overrightarrow{c} = [c_{e,1}; c_{e,2}; ...c_{s,1}; c_{s,2}; ...] \tag{6}$$

At each step t of the decoder, we first update the states of the decoder LSTM:

$$h_{\text{dec},t} = \text{LSTM}(\overrightarrow{c}_{\hat{q}_{t-1}}, h_{\text{dec},t-1}) \tag{7}$$

Finally, we attend over the document representations given the current decoder state using dot-product attention (Bahdanau et al., 2015):

$$\hat{a}_t = \text{softmax}(h_{\text{dec},t} \overrightarrow{h}_{\text{enc}}^{\intercal}) \tag{8}$$

$$v_t = \sum_i \hat{a}_{t,i} \overrightarrow{h}_{\text{enc},i} \tag{9}$$

The score for the ith candidate $\overrightarrow{c}_i$ is

$$o_t = \hat{W}[h_{\text{dec},t}; v_t] + \hat{b} \tag{10}$$

$$s_{t,i} = o_t \overrightarrow{c}_i^{\intercal} \tag{11}$$

$$\hat{q}_t = \text{argmax}(s_t) \tag{12}$$

Value-generation. The pervious template decoder produces logical form template $\hat{q}$, which is not executable because it does not include utterance candidates c_u. To generate full-specified executable logical forms q, we use a separate value pointer-decoder that selects among utterance tokens. The attention input for this decoder is identical to that of the template decoder. The pointer candidates c_u are obtained by running a separate BERT encoder on the utterance u. The produced values are inserted into each slot in $\hat{q}$ to form q.

Both template and value decoders are trained using cross-entropy loss with respect to the ground-truth sequence of candidates.

2.3 Backward utterance generator

The utterance generator G produces an utterance $u = G(q, e)$ for the logical form q in the environment e. The alignment problem between q and the environment description w is simpler than that between u and w because environment candidates c_e (e.g. column names) in q are described by corresponding spans in w (e.g. column schemas in database schema). To leverage this deterministic alignment, we augment c_e in q with relevant spans from w, and encode this augmented logical form $\tilde{q}$. The pointer-decoder selects among words c_v from a fixed vocabulary (e.g. when, where, who) and words $c_{\tilde{q}}$ from $\tilde{q}$. Figure 3 illustrates the backward utterance generator.

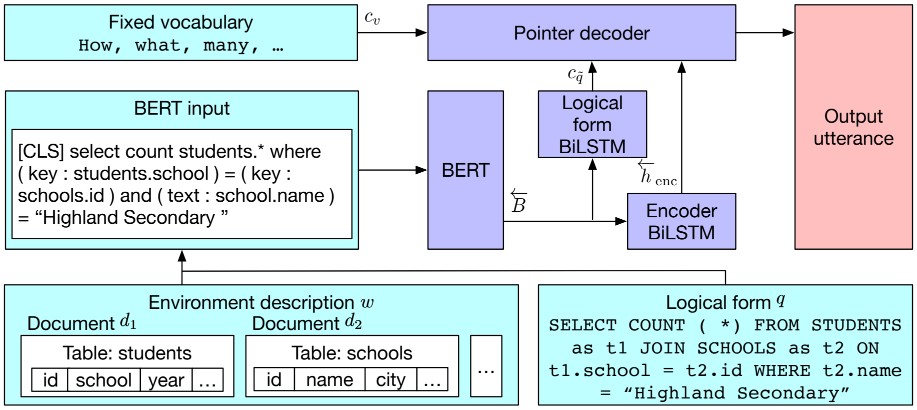

Fixed vocabulary
How, what, many, …
c_v
Pointer decoder
$c_{\tilde{q}}$
Logical form BiLSTM
BERT input
[CLS] select count students.* where (key : students.school) = (key : schools.id) and (text : school.name) = "Highland Secondary "
BERT
$\overleftarrow{B}$
$\overleftarrow{h}_{\text{enc}}$
Encoder BiLSTM
Output utterance
Environment description w
Document d_1
Table: students
id
school
year
…
Document d_2
Table: schools
id
name
city
…
…
Logical form q
SELECT COUNT (*) FROM STUDENTS as t1 JOIN SCHOOLS as t2 ON t1.school = t2.id WHERE t2.name = "Highland Secondary"

Figure 3: Backward utterance generator. Model components are shown in purple, inputs in blue, and outputs in red. First, we encode the input logical form along with environment description for each of its symbols. we subsequently encode using LSTMs to form the input and environment-level candidate token representations. A pointer-decoder attends over the input and selects among candidate representations to produce the output utterance.

First, we encode the logical form using BERT.

$$\overleftarrow{B} = \text{BERT}_{\leftarrow}(\tilde{q}) \tag{13}$$

Next, we apply a bidirectional LSTM to obtain the input encoding $\overleftarrow{h}_{\text{enc}}$ and another bidirectional LSTM to obtain representations of tokens in the augmented logical form $c_{\tilde{q}}$.

$$\begin{aligned} \overleftarrow{h}_{\text{enc}} &= \text{BiLSTM}(\overleftarrow{B}) && (14) \\ c_{\tilde{q}} &= \text{BiLSTM}(\overleftarrow{B}) && (15) \end{aligned}$$

To represent c_v, we use word embeddings from $\text{BERT}_{\leftarrow}$. Finally, we apply a pointer-decoder that attends over $\overleftarrow{h}_{\text{enc}}$ and selects among candidates $\overleftarrow{c} = [c_{\tilde{q}}; c_v]$ to obtain the predicted utterance.

2.4 Synthesizing cycle-consistent examples

Having trained a forward semantic parser F and a backward utterance generator G in environment e, we can synthesize new examples with which to adapt the parser in the new environment e'. First, we sample a logical form q using a grammar (Algorithm 1 in Section 2.1). Next, we predict an utterance $u' = G(q, e')$. Because G was trained only on e, many of its outputs are low-quality or do not correspond to its input q. On their own, these examples (u', q) do not facilitate parser adaptation (see Section 3.1 for analyses).

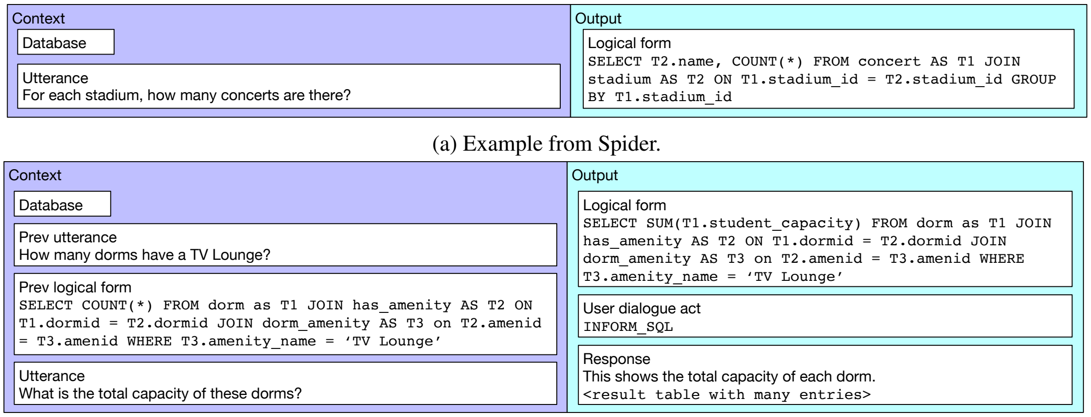
To filter out low-quality examples, we additionally predict a logical form $q' = F(u', e')$, and keep only examples that are **cycle consistent** — the synthesized logical form q' is equivalent to the originally sampled logical form q in e'. In the case of SQL parsing, the example is cycle-consistent if executing the synthesized query $\textsc{Exe}(q', e')$ results in the same denotation (i.e. same set of database records) as executing the original sampled query $\textsc{Exe}(q, e')$. Finally, we combine cycle-consistent examples synthesized in e' with the original training data in e to retrain and adapt the parser.

3 Experiments

We evaluate performance on the Spider (Yu et al., 2018b), Sparc (Yu et al., 2019b), and CoSQL (Yu et al., 2019a) zero-shot semantic parsing tasks. Table 1 shows dataset statistics. Figure 4 shows examples from each dataset. For all three datasets, we use preprocessing steps from Zhang et al. (2019) to preprocess SQL logical forms. Evaluation consists of **exact match over logical form templates** (EM) in which values are stripped out, as well as **execution accuracy** (EX). Official evaluations also recently incorporated **fuzz-test accuracy** (FX) as tighter variant of execution accuracy. In fuzz-testing, the query is executed over randomized database content numerous times. Compared to an execution match, a fuzz-test execution match is less likely to be spurious (e.g. the predicted query coincidentally executes to the correct result). FX implementation is not public as of writing, hence we only report test FX.

Spider. Spider is a collection of database-utterance-SQL query triplets. The task involves producing the SQL query given the utterance and the database. Figure 2 and 3 show preprocessed input for the parser and generator.

Sparc. In Sparc, the user repeatedly asks questions that must be converted to SQL queries



Context

Database

Utterance
For each stadium, how many concerts are there?

Output

Logical form

```
SELECT T2.name, COUNT(*) FROM concert AS T1 JOIN
stadium AS T2 ON T1.stadium_id = T2.stadium_id GROUP
BY T1.stadium_id
```

(a) Example from Spider.

Context

Database

Prev utterance
How many dorms have a TV Lounge?

Prev logical form

```
SELECT COUNT(*) FROM dorm as T1 JOIN has_amenity AS T2 ON
T1.dormid = T2.dormid JOIN dorm_amenity AS T3 on T2.amenid
= T3.amenid WHERE T3.amenity_name = 'TV Lounge'
```

Utterance
What is the total capacity of these dorms?

Output

Logical form

```
SELECT SUM(T1.student_capacity) FROM dorm as T1 JOIN
has_amenity AS T2 ON T1.dormid = T2.dormid JOIN
dorm_amenity AS T3 on T2.amenid = T3.amenid WHERE
T3.amenity_name = 'TV Lounge'
```

User dialogue act
`INFORM_SQL`

Response
This shows the total capacity of each dorm.
`<result table with many entries>`

(b) Example from CoSQL.

Figure 4: Examples from (a) Spider and (b) CoSQL. Context and output are respectively shown in purple and blue. We do not show Sparc because its data format is similar to CoSQL, but without user dialogue act prediction and without response generation. For our experiments, we produce the output logical form given the data, utterance, and the previous logical form if applicable. During evaluation, the previous logical form is the output of the model during the previous turn (i.e. no teacher forcing on ground-truth previous output).

	Spider	Sparc	CoSQL
# database	200	200	200
# tables	1020	1020	1020
# utterances	10,181	4298	3007
# logical forms	5,693	12,726	15,598
multi-turn	no	yes	yes

Table 1: Dataset statistics.

by the system. Compared to Spider, Sparc additionally contains prior interactions from the same user session (e.g. database-utterance-query-previous query quadruplets). For Sparc evaluation, we concatenate the previous system-produced query (if present) to each utterance. For example, suppose the system was previously asked "where is Tesla born?" and is now asked "how many people are born there?", we produce the utterance `[PREV] SELECT birth_place FROM people WHERE name = 'Tesla' [UTT] how many people are born there ?` For training and data synthesis, the ground-truth previous query is used as generation context for forward parsing and backward utterance generation.

CoSQL. CoSQL is combines task-oriented dialogue and semantic parsing. It consists of a number of tasks, such as response generation, user act prediction, and state-tracking. We focus on state-tracking, in which the user intent is mapped to a SQL query. Similar to Zhang et al. (2019), we restrict the context to be the previous query and the current utterance. Hence, the input utterance and environment description are obtained in the same way as that used for Sparc.

3.1 Results

We primarily compare GAZP with the baseline forward semantic parser, because prior systems produce queries without values which are not executable. We include one such non-executable model, EditSQL (Zhang et al., 2019), one of the top parsers on Spider at the time of writing, for reference. However, EditSQL EM is not directly comparable because of different outputs.

Due to high variance from small datasets, we tune the forward parser and backward generator using cross-validation. We then retrain the model with early stopping on the development set using hyperparameters found via cross-validation. For each task, we synthesize 100k examples, of which ∼40k are kept after checking for cycle-consistency. The adapted parser is trained using the same hyperparameters as the baseline. Please see appendix A.2 for hyperparameter settings. Appendix A.3 shows examples of synthesized adaptation examples and compares them to real examples.

Table 2 shows that adaptation by GAZP results in consistent performance improvement across Spider, Sparc, and CoSQL in terms of EM, EX, and FX. We also examine the performance breakdown across query classes and turns (details in appendix A.4). First, we divide queries into difficulty classes based on the number of SQL components, selections, and conditions (Yu et al., 2018b). For example, queries that contain more components such as `GROUP`, `ORDER`, `INTERSECT`,

Model	Spider					Sparc					CoSQL				
	dev		test			dev		test			dev		test		
	EM	EX	EM	EX	FX	EM	EX	EM	EX	FX	EM	EX	EM	EX	FX
EditSQL	57.6	n/a	53.4	n/a	n/a	47.2	n/a	47.9	n/a	n/a	39.9	n/a	40.8	n/a	n/a
Baseline	56.8	55.4	52.1	49.8	51.1	46.4	44.0	45.9	43.5	42.8	39.3	36.6	37.2	34.9	33.8
GAZP	**59.1**	**59.2**	**53.3**	**53.5**	**51.7**	**48.9**	**47.8**	45.9	**44.6**	**43.9**	**42.0**	**38.8**	**39.7**	**35.9**	**36.3**

Table 2: Development set evaluation results on Spider, Sparc, and CoSQL. **EM** is exact match accuracy of logical form templates without values. **EX** is execution accuracy of fully-specified logical forms with values. **FX** is execution accuracy from fuzz-testing with randomized databases. **Baseline** is the forward parser without adaptation. **EditSQL** is a state-of-the-art language-to-SQL parser that produces logical form templates that are not executable.

Model	Spider			Sparc			CoSQL		
	EM	EX	# syn	EM	EX	# syn	EM	EX	# syn
Baseline	56.8	55.4	40557	46.4	44.0	45221	39.3	36.6	33559
GAZP	59.1	**59.2**	40557	**48.9**	**47.8**	45221	**42.0**	**38.8**	33559
nocycle	55.6	52.3	97655	41.1	40.0	81623	30.7	30.8	78428
syntrain	54.8	52.1	39721	47.4	45.2	44294	38.7	34.3	31894
EM consistency	**61.6**	56.9	35501	48.4	45.9	43521	41.9	37.7	31137

Table 3: Ablation performance on development sets. For each one, 100,000 examples are synthesized, out of which queries that do not execute or execute to the empty set are discarded. "nocycle" uses adaptation without cycle-consistency. "syntrain" uses data-augmentation on training environments. "EM consistency" enforces logical form instead of execution consistency.

nested subqueries, column selections, and aggregators, etc are considered to be harder. Second, we divide multi-turn queries into how many turns into the interaction they occur for Sparc and CoSQL (Yu et al., 2019b,a). We observe that the gains in GAZP are generally more pronounced in more difficult queries and in turns later in the interaction. Finally, we answer the following questions regarding the effectiveness of cycle-consistency and grounded adaptation.

Does adaptation on inference environment outperform data-augmentation on training environment? For this experiment, we synthesize data on training environments instead of inference environments. The resulting data is similar to data augmentation with verification. As shown in the "syntrain" row of Table 3, retraining the model on the combination of this data and the supervised data leads to overfitting in the training environments. A method related to data-augmentation is jointly supervising the model using the training data in the reverse direction, for example by generating utterance from query (Fried et al., 2018; Cao et al., 2019). For Spider, we find that this dual objective (57.2 EM) underperforms GAZP adaptation (59.1 EM). Our results indicate that adaptation to the new environment significantly outperforms augmentation in the training environment.

How important is cycle-consistency? For this experiment, we do not check for cycle-consistency and instead keep all synthesized queries in the inference environments. As shown in the "nocycle" row of Table 3, the inclusion of cycle-consistency effectively prunes ∼60% of synthesized examples, which otherwise significantly degrade performance. This shows that enforcing cycle-consistency is crucial to successful adaptation.

In another experiment, we keep examples that have consistent logical forms, as deemed by string match (e.g. $q == q'$), instead of consistent denotation from execution. The "EM consistency" row of Table 3 shows that this variant of cycle-consistency also improves performance. In particular, EM consistency performs similarly to execution consistency, albeit typically with lower execution accuracy.

How much GAZP synthesized data should one use for grounded adaptation? For this experiment, we vary the amount of cycle-consistent syn-

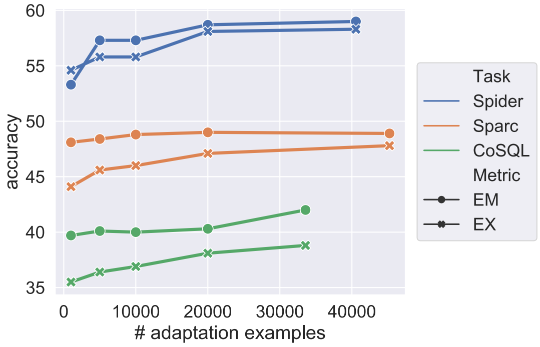


Figure 5: Effect of amount of synthesized data on adaptation performance on the development set. EM and EX denote template exact match and logical form execution accuracy, respectively. The x-axis shows the number of cycle-consistent examples synthesized in the inference environments (e.g. all databases in the development set).

thesized data used for adaptation. Figure 5 shows that that adaptation performance generally increases with the amount of synthesized data in the inference environment, with diminishing return after 30-40k examples.

4 Related work

Semantic parsing. Semantic parsers parse natural language utterances into executable logical forms with respect to an environment (Zelle and Mooney, 1996; Zettlemoyer and Collins, 2005; Liang et al., 2011). In zero-shot semantic parsing, the model is required to generalize to environments (e.g. new domains, new database schemas) not seen during training (Pasupat and Liang, 2015; Zhong et al., 2017; Yu et al., 2018b). For language-to-SQL zero-shot semantic parsing, a variety of methods have been proposed to generalize to new databases by selecting from table schemas in the new database (Zhang et al., 2019; Guo et al., 2019). Our method is complementary to these work — the synthesis, cycle-consistency, and adaptation steps in GAZP can be applied to any parser, so long as we can learn a backward utterance generator and evaluate logical-form equivalence.

Data augmentation. Data augmentation transforms original training data to synthesize artificial training data. Krizhevsky et al. (2017) crop and rotate input images to improve object recognition. Dong et al. (2017) and Yu et al. (2018a) respectively paraphrase and back-translate (Sennrich et al., 2016; Edunov et al., 2018) questions and documents to improve question-answering. Jia and Liang (2016) perform data-recombination in the training domain to improve semantic parsing. Hannun et al. (2014) superimpose noisy background tracks with input tracks to improve speech recognition. Our method is distinct from data-augmentation in the following ways. First, we synthesize data on logical forms sampled from the new environment instead of the original environment, which allows for adaptation to the new environments. Second, we propose cycle-consistency to prune low-quality data and keep high-quality data for adaptation. Our analyses show that these core differences from data-augmentation are central to improving parsing performance.

Cycle-consistent generative adversarial models (cycle-GANs). In cycle-GAN (Zhu et al., 2017; Hoffman et al., 2018), a generator forms images that fools a discriminator while the discriminator tries distinguish generated images from naturally occurring images. The the adversarial objectives of the generator and the discriminator are optimized jointly. Our method is different from cycle-GANs in that we do not use adversarial objectives and instead rely on matching denotations from executing synthesized queries. This provides an exact signal compared to potentially incorrect outputs by the discriminator. Morevoer, cycle-GANs only synthesize the input and verify whether the input is synthesized (e.g. the utterance looks like a user request). In contrast, GAZP synthesizes both the input and the output, and verifies consistency between the input and the output (e.g. the utterance matches the query).

5 Conclusion and Future work

We proposed GAZP to adapt an existing semantic parser to new environments by synthesizing cycle-consistent data. GAZP improved parsing performance on three zero-shot parsing tasks. Our analyses showed that GAZP outperforms data augmentation, performance improvement scales with the amount of GAZP-synthesized data, and cycle-consistency is central to successful adaptation.

In principle, GAZP applies to any problems that lack annotated data and differ between training and inference environments. One such area is robotics, where one trains in simulation because it is prohibitively expensive to collect annotated trajectories in the real world. In future work, we will consider how to interpret environment specifications to facilitate grounded adaptation in these other areas.

Acknowledgement

This research was supported in part by the ARO (W911NF-16-1-0121) and the NSF (IIS-1252835, IIS-1562364). We thank Julian Michael for helpful discussions, and our reviewers for engaging and constructive feedback. We also thank Bo Pang and Tao Yu for helping us with running the official evaluations.

References

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. *ICLR*.

Ruisheng Cao, Su Zhu, Chen Liu, Jieyu Li, and Kai Yu. 2019. Semantic parsing with dual learning. In *ACL*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*.

Li Dong, Jonathan Mallinson, Siva Reddy, and Mirella Lapata. 2017. Learning to paraphrase for question answering. In *EMNLP*.

Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. 2018. Understanding back-translation at scale. In *EMNLP*.

Daniel Fried, Ronghang Hu, Volkan Cirik, Anna Rohrbach, Jacob Andreas, Louis-Philippe Morency, Taylor Berg-Kirkpatrick, Kate Saenko, Dan Klein, and Trevor Darrell. 2018. Speaker-follower models for vision-and-language navigation. In *NeurIPS*.

Arthur C Graesser, Patrick Chipman, Brian C Haynes, and Andrew Olney. 2005. AutoTutor: An intelligent tutoring system with mixed-initiative dialogue. *IEEE Transactions on Education*.

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-SQL in cross-domain database with intermediate representation. In *ACL*.

Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. 2014. Deep Speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*.

Judy Hoffman, Eric Tzeng, Taesung Park, Jun-Yan Zhu, Phillip Isola, Kate Saenko, Alexei A. Efros, and Trevor Darrell. 2018. CyCADA: Cycle consistent adversarial domain adaptation. In *ICML*.

Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. In *ACL*.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *ICLR*.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*.

Percy Liang, Michael I. Jordan, and Dan Klein. 2011. Learning dependency-based compositional semantics. *Computational Linguistics*.

Cynthia Matuszek, Evan Herbst, Luke Zettlemoyer, and Dieter Fox. 2013. Learning to parse natural language commands to a robot control system. *Experimental Robotics*.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. In *ACL*.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2020. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. In *NeurIPS Workshop on Energy Efficient Machine Learning and Cognitive Computing*.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Improving neural machine translation models with monolingual data. In *ACL*.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *NIPS*.

Wen-tau Yih, Xiaodong He, and Christopher Meek. 2014. Semantic parsing for single-relation question answering. In *ACL*.

Adams Wei Yu, David Dohan, Minh-Thang Luong, Rui Zhao, Kai Chen, Mohammad Norouzi, and Quoc V Le. 2018a. Qanet: Combining local convolution with global self-attention for reading comprehension. In *ICLR*.

Tao Yu, Rui Zhang, He Yang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, Vincent Zhang, Caiming Xiong, Richard Socher, Walter Lasecki, and Dragomir Radev. 2019a. Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases. In *EMNLP*.

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. 2018b. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *EMNLP*.

Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, Emily Ji, Shreya Dixit, David Proctor, Sungrok Shim, Jonathan Kraft, Vincent Zhang, Caiming Xiong, Richard Socher, and Dragomir Radev. 2019b. Sparc: Cross-domain semantic parsing in context. In *ACL*.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *AAAI/IAAI*.

Luke S. Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI*.

Rui Zhang, Tao Yu, He Yang Er, Sungrok Shim, Eric Xue, Xi Victoria Lin, Tianze Shi, Caiming Xiong, Richard Socher, and Dragomir Radev. 2019. Editing-based sql query generation for cross-domain context-dependent questions. In *EMNLP*.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.

Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. 2017. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *ICCV*.

A Appendix

A.1 Coverage and multi-turn sampling

When we build an empirical distribution over templates on the training set of Spider, we observe a 85% coverage of dev set templates. That is, 85% of dev set examples have a query whose template occurs in the training set. In other words, while this simple template-filling sampling scheme doesn't provide full coverage over the dev set as a complex grammar would, it covers a large portion of examples.

For Sparc and CoSQL, the sampling procedure is similar to Algorithm 1. However, because there are two queries (one previous, one current), we first sample a previous query z_1' from $P_{\text{temp}}(z)$, then sample the current query z_2' from $P_{\text{temp}}(z|z_1')$. As before, the empirical template distributions are obtained by counting templates in the training set.

A.2 Hyperparameters

Dropout location	Forward parser		
	Spider	Sparc	CoSQL
post-BERT	0.1	0.1	0.1
post-enc LSTMs	0.1	0.3	0.1
pre-dec scorer	0.1	0.1	0.3

Table 4: Dropout rates for the forward parser.

Dropout location	Backward generator		
	Spider	Sparc	CoSQL
post-BERT	0.1	0.3	0.1
post-enc LSTMs	0.1	0.1	0.1
pre-dec scorer	0.1	0.1	0.3

Table 5: Dropout rates for the backward generator.

We use 300-dimensional LSTMs throughout the model. The BERT model we use is DistilBERT (Sanh et al., 2020), which we optimize with Adam (Kingma and Ba, 2015) with an initial learning rate of $5e-5$. We train for 50 epochs with a batch size of 10 and gradient clipping with a norm of 20. We use dropout after BERT, after encoder LSTMs, and before the pointer scorer. The values for these dropouts used by our leaderboard submissions are shown in Table 4 and Table 5. For each task, these rates are tuned using 3-fold cross-validation with a coarse grid-search over values $\{0.1, 0.3\}$ for each dropout with a fixed seed.

A single training run of the forward parser took approximately 16 hours to run on a single NVIDIA Titan X GPU. Each task required 3 folds in addition to the final official train/dev run. For each fold, we grid-searched over dropout rates, which amounts to 8 runs. In total, we conducted 27 runs on a Slurm cluster. Including pretrained BERT parameters, the final forward parser contains 142 million parameters. The final backward utterance generator contains 73 million parameters.

list all the last name of owners in alphabetical order .	`select last_name from Owners order by last_name`
how many friend are there ?	`select count ( * ) from Friend`
what is the id of the votes that has been most distinct contestants ?	`"select T2.vote_id from CONTESTANTS as T1 join VOTES as T2 on T1.contestant_number = T2.contestant_number group by ( T2.vote_id ) order by count ( T1.contestant_number ) desc limit 1`
what are the name of higher ?	`select name from Highschooler`
how many car makers has the horsepower of 81 ?	`select count ( * ) from cars_data as T1 join car_names as T2 on T1.Id = T2.MakeId join model_list as T3 on T2.Model = T3.Model join car_makers as T4 on T3.Maker = T4.Id where T1.Horsepower = ’81’`
what are the starts of hiring who are located in the city of Bristol ?	`select T2.Start_from from employee as T1 join hiring as T2 on T1.Employee_ID = T2.Employee_ID where T1.City = ’Bristol’`
find the name and district of the employee that has the highest evaluation bonus .	`select T2.Name , T4.District from evaluation as T1 join employee as T2 on T1.Employee_ID = T2.Employee_ID join hiring as T3 on T2.Employee_ID = T3.Employee_ID join shop as T4 on T3.Shop_ID = T4.Shop_ID order by T1.Bonus desc limit 1`
what is the cell number of the owners with the largest charges amount ?	`select T1.cell_number from Owners as T1 join Charges as T2 order by T2.charge_amount desc limit 1`
what is the minimum , average , and maximum grade of all high schooler ?	`select min ( grade ) , avg ( grade ) , max ( grade ) from Highschooler`
what is the age of the teacher who has the most course ?	`select T1.Age from teacher as T1 join course_arrange as T2 on T1.Teacher_ID = T2.Teacher_ID group by T2.Teacher_ID order by sum ( T2.Grade ) desc limit 1`

Table 6: Examples of synthesized queries

A.3 Synthesized examples

In order to quantify the distribution of synthesized examples, we classify synthesized queries according to the difficulty criteria from Spider (Yu et al., 2018b). Compared to the Spider development set, GAZP-synthesized data has an average of 0.60 vs. 0.47 joins, 1.21 vs. 1.37 conditions, 0.20 vs. 0.26 group by's, 0.23 vs. 0.25 order by's, 0.07 vs. 0.04 intersections, and 1.25 vs. 1.32 selection columns per query. This suggests that GAZP queries are similar to real data.

Moreover, we example a random sample of 60 synthesized examples. Out of the 60, 51 are correct. Mistakes come from aggregation over wrong columns (e.g. "has the most course" becomes `order by sum T2.grade`) and underspecification (e.g. "lowest of the stadium who has the lowest age"). There are grammatical errors (e.g. "that has the most" becomes "that has been most"), but most questions are fluent and sensible (e.g. "find the name and district of the employee that has the highest evaluation bonus"). A subset of these queries are shown in Table 6.

A.4 Performance breakdown

		easy	medium	hard	extra	all
count		470	857	463	357	2147
baseline	EM	**75.3**	54.9	45.0	**24.8**	52.1
	EX	**60.3**	52.7	47.5	32.6	49.8
	FX	**73.6**	52.9	44.8	**26.4**	51.1
GAZP	EM	73.1	**58.7**	**47.2**	23.3	**53.3**
	EX	59.6	**59.2**	**52.3**	**33.3**	**53.5**
	FX	71.9	**55.3**	**46.1**	24.5	**51.7**

Table 7: Difficulty breakdown for Spider test set.

		easy	medium	hard	extra	all
count		993	845	399	261	2498
baseline	EM	**68.9**	36.9	31.2	11.1	45.9
	EX	**61.9**	35.6	30.6	18.8	43.5
	FX	**65.9**	32.5	**28.1**	10.7	42.8
GAZP	EM	66.5	**39.6**	**38.4**	**14.2**	45.9
	EX	60.1	**39.5**	**31.1**	**20.3**	**44.6**
	FX	65.3	**36.8**	26.3	**12.6**	**43.9**

Table 8: Difficulty breakdown for Sparc test set.

		easy	medium	hard	extra	all
count		730	607	358	209	1904
baseline	EM	58.2	28.0	20.6	**18.8**	37.2
	EX	47.1	27.2	26.8	**28.2**	34.9
	FX	51.9	24.1	21.2	**20.6**	33.8
GAZP	EM	**60.0**	**33.8**	**23.1**	13.9	**39.7**
	EX	**48.1**	**28.3**	**41.0**	23.9	**35.9**
	FX	**55.1**	**26.9**	**25.7**	16.7	**36.3**

Table 9: Difficulty breakdown for CoSQL test set.

		turn 1	turn 2	turn 3	turn 4+
count		842	841	613	202
baseline	EM	**69.9**	41.8	28.9	16.4
	EX	**67.8**	36.9	28.1	16.9
	FX	**70.2**	35.7	24.8	13.4
GAZP	EM	67.8	41.9	**29.7**	**19.6**
	EX	66.3	**40.1**	**29.0**	**19.8**
	FX	68.8	**38.3**	**25.9**	**18.3**

Table 10: Turn breakdown for Sparc test set

In addition to the main experiment results in Table 2 of Section 3.1, we also examine the performance breakdown across query classes and turns.

GAZP improves performance on harder queries. First, we divide queries into difficulty classes following the classification in Yu et al. (2018b). These difficulty classes are based on the number of SQL components, selections, and conditions. For example, queries that contain more SQL keywords such as `GROUP BY`, `ORDER BY`, `INTERSECT`, nested subqueries, column selections, and aggregators, etc are considered to be harder. Yu et al. (2018b) shows examples of SQL queries in the four hardness categories. Note that **extra** is a catch-all category for queries that exceed qualifications of **hard**, as a result it includes artifacts (e.g. set exclusion operations) that may introduce other confounding factors. Tables 7, 8, and 9 respectively break down the performance of models on Spider, Sparc, and CoSQL. We observe that the gains in GAZP are generally more pronounced in more difficult queries. This finding is consistent across tasks (with some variance) and across three evaluation metrics.

One potential explanation for this gain is that the generalization problem is exacerbated in more

		turn 1	turn 2	turn 3	turn 4+
count		548	533	372	351
baseline	EM	47.3	36.5	32.3	28.5
	EX	43.8	**34.3**	30.3	27.9
	FX	46.2	31.9	29.4	23.4
GAZP	EM	**50.0**	36.7	**35.7**	**30.3**
	EX	**46.4**	32.3	**32.2**	**30.2**
	FX	**50.0**	**32.8**	**31.4**	**27.1**

Table 11: Turn breakdown for CoSQL test set.

difficult queries. Consider the example of language-to-SQL parsing, in which we have trained a parser on an university database and are now evaluating it on a sales database. While it is difficult to produce simple queries in the sales database due to ta lack of training data, it is likely even more difficult to produce nested queries, queries with groupings, queries with multiple conditions, etc. Because GAZP synthesizes queries — including difficult ones — in the sales database, the adapted parser learns to handle these cases. In contrast, simpler queries are likely easier to learn, hence adaptation does not help as much.

GAZP improves performance in longer interactions. For Sparc and CoSQL, which include multi-turn interactions between the user and the system, we divide queries into how many turns into the interaction they occur. This classification in described in Yu et al. (2019b) and Yu et al. (2019a). Tables 10 and 11 respectively break down the performance of models on Sparc and CoSQL. We observe that the gains in GAZP are more pronounced in turns later in the interaction. Against, this finding is consistent not only across tasks, but across the three evaluation metrics.

A possible reason for this gain is that the conditional sampling procedure shown in Algorithm 1 improves multi-turn parsing by synthesizing multi-turn examples. How much additional variation should we expect in a multi-turn setting? Suppose we discover T coarse-grain templates by counting the training data, where each coarse-grain template has S slots on average. For simplicity, let us ignore value slots and only consider column slots. Given a new database with N columns, the number of possible filled queries is on the order of $O\left(T \times \binom{S}{N}\right)$. For K turns, the number of possible queries sequences is then $O\left(\left(T \times \binom{S}{N}\right)^{K}\right)$. This exponential increase in query variety may improve parser performance on later-turn queries (e.g. those with a previous interaction), which in turn reduce cascading errors throughout the interaction.