SmBoP: Semi-autoregressive Bottom-up Semantic Parsing

Ohad Rubin
Tel Aviv University
ohadr@mail.tau.ac.il

Jonathan Berant
Tel Aviv University
Allen Institute for AI
joberant@cs.tau.ac.il

Abstract

The de-facto standard decoding method for semantic parsing in recent years has been to autoregressively decode the abstract syntax tree of the target program using a top-down depth-first traversal. In this work, we propose an alternative approach: a Semi-autoregressive Bottom-up Parser (SmBoP) that constructs at decoding step t the top-K sub-trees of height $\leq t$. Our parser enjoys several benefits compared to top-down autoregressive parsing. From an efficiency perspective, bottom-up parsing allows to decode all sub-trees of a certain height in parallel, leading to logarithmic runtime complexity rather than linear. From a modeling perspective, a bottom-up parser learns representations for meaningful semantic sub-programs at each step, rather than for semantically-vacuous partial trees. We apply SmBoP on Spider, a challenging zero-shot semantic parsing benchmark, and show that SmBoP leads to a 2.2x speed-up in decoding time and a ~5x speed-up in training time, compared to a semantic parser that uses autoregressive decoding. SmBoP obtains 71.1 denotation accuracy on Spider, establishing a new state-of-the-art, and 69.5 exact match, comparable to the 69.6 exact match of the autoregressive RAT-SQL+Grappa.

1 Introduction

Semantic parsing, the task of mapping natural language utterances into programs (Zelle and Mooney, 1996; Zettlemoyer and Collins, 2005; Clarke et al.; Liang et al., 2011), has converged in recent years on a standard encoder-decoder architecture. Recently, meaningful advances emerged on the encoder side, including developments in Transformer-based architectures (Wang et al., 2020a) and new pretraining techniques (Yin et al., 2020; Herzig et al., 2020; Yu et al., 2020; Deng et al., 2020; Shi et al., 2021). Conversely, the decoder has remained roughly constant for years, where the abstract syntax tree of the target program is autoregressively decoded in a top-down manner (Yin and Neubig, 2017; Krishnamurthy et al., 2017; Rabinovich et al., 2017).

Bottom-up decoding in semantic parsing has received little attention (Cheng et al., 2019; Odena et al., 2020). In this work, we propose a bottom-up semantic parser, and demonstrate that equipped with recent developments in Transformer-based (Vaswani et al., 2017) architectures, it offers several advantages. From an efficiency perspective, bottom-up parsing can naturally be done *semi-autoregressively*: at each decoding step t, the parser generates *in parallel* the top-K program sub-trees of depth $\leq t$ (akin to beam search). This leads to runtime complexity that is logarithmic in the tree size, rather than linear, contributing to the rocketing interest in efficient and greener artificial intelligence technologies (Schwartz et al., 2020). From a modeling perspective, neural bottom-up parsing provides learned representations for meaningful (and executable) sub-programs, which are sub-trees computed during the search procedure, in contrast to top-down parsing, where hidden states represent partial trees without clear semantics.

Figure 1 illustrates a single decoding step of our parser. Given a beam Z_t with $K = 4$ trees of height t (blue vectors), we use *cross-attention* to contextualize the trees with information from the input question (orange). Then, we score the *frontier*, that is, the set of all trees of height $t + 1$ that can be constructed using a grammar from the current beam, and the top-K trees are kept (purple). Last, a representation for each of the new K trees is generated and placed in the new beam Z_{t+1}. After T decoding steps, the parser returns the highest-scoring tree in Z_T that corresponds to a full program. Because we have gold trees at training time, the entire model is trained jointly using maximum likelihood.

We evaluate our model, SmBoP[1] (SeMi-autoregressive Bottom-up semantic Parser), on Spi-

[1] Rhymes with 'MMMBop'.

Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 311–324
June 6–11, 2021. ©2021 Association for Computational Linguistics

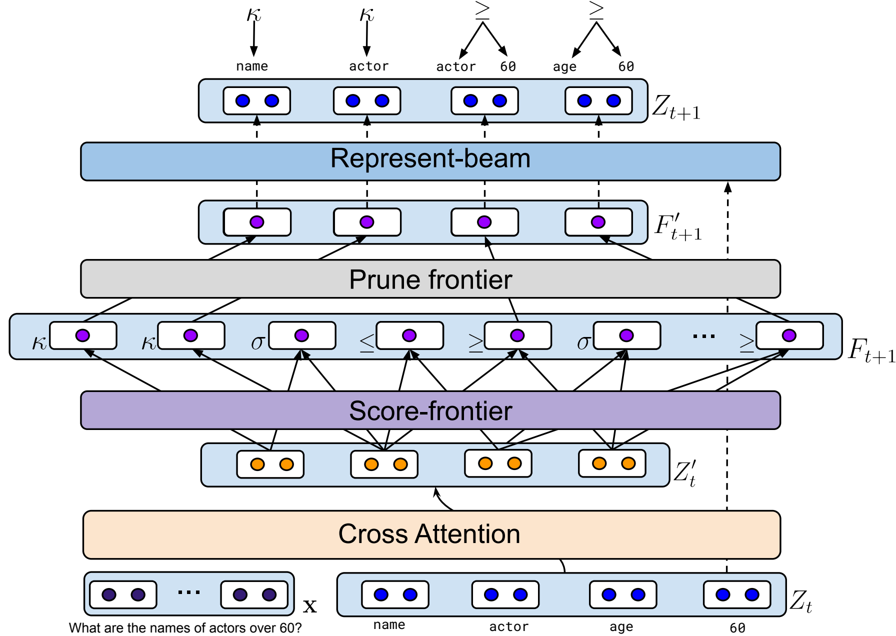


Figure 1: An overview of the decoding procedure of SMBOP. Z_t is is the beam at step t, Z'_t is the contextualized beam after cross-attention, F_{t+1} is the frontier ($\kappa, \sigma, \geq$ are logical operations applied on trees, as explained below), F'_{t+1} is the pruned frontier, and Z_{t+1} is the new beam. At the top we see the new trees created in this step. For $t = 0$ (depicted here), the beam contains the predicted schema constants and DB values.

DER (Yu et al., 2018), a challenging zero-shot text-to-SQL dataset. We implement the RAT-SQL+GRAPPA encoder (Yu et al., 2020), currently the best model on SPIDER, and replace the autoregressive decoder with the semi-autoregressive SMBOP. SMBOP obtains an exact match accuracy of 69.5, comparable to the autoregressive RAT-SQL+GRAPPA at 69.6 exact match, and to current state-of-the-art at 69.8 exact match (Zhao et al., 2021), which applies additional pretraining. Moreover, SMBOP substantially improves state-of-the-art in denotation accuracy, improving performance from 68.3 $\rightarrow$ 71.1. Importantly, compared to autoregressive semantic parsing , we observe an average speed-up of 2.2x in decoding time, where for long SQL queries, speed-up is between 5x-6x, and a training speed-up of ~5x.[2]

2 Background

Problem definition We focus in this work on text-to-SQL semantic parsing. Given a training set $\{(x^{(i)}, y^{(i)}, S^{(i)})\}_{i=1}^N$, where $x^{(i)}$ is an utterance, $y^{(i)}$ is its translation to a SQL query, and $S^{(i)}$ is the schema of the target database (DB), our goal is to learn a model that maps new question-schema pairs (x, S) to the correct SQL query y. A DB schema S includes : (a) a set of tables, (b) a set of columns for each table, and (c) a set of foreign key-primary key column pairs describing relations between table columns. Schema tables and columns are termed schema constants, and denoted by $\mathcal{S}$.

RAT-SQL encoder This work is focused on decoding, and thus we implement the state-of-the-art RAT-SQL encoder (Wang et al., 2020b), on top of GRAPPA (Yu et al., 2020), a pre-trained encoder for semantic parsing. We now briefly review this encoder for completeness.

The RAT-SQL encoder is based on two main ideas. First, it provides a joint contextualized representation of the utterance and schema. Specifically, the utterance x is concatenated to a linearized form of the schema S, and they are passed through a stack of Transformer (Vaswani et al., 2017) layers. Then, tokens that correspond to a single schema constant are aggregated, which results in a final contextualized representation $(\mathbf{x}, \mathbf{s}) = (\mathbf{x}_1, \dots, \mathbf{x}_{|x|}, \mathbf{s}_1, \dots, \mathbf{s}_{|\mathbf{s}|})$, where $\mathbf{s}_i$ is a vector representing a single schema constant. This contextualization of x and S leads to better representation and alignment between the utterance and schema.

Second, RAT-SQL uses *relational-aware self-*

[2]Our code is available at https://github.com/OhadRubin/SmBop

attention (Shaw et al., 2018) to encode the structure of the schema and other prior knowledge on relations between encoded tokens. Specifically, given a sequence of token representations $(\mathbf{u}_1, \ldots, \mathbf{u}_{|\mathbf{u}|})$, relational-aware self-attention computes a scalar similarity score between pairs of token representations $e_{ij} \propto \mathbf{u}_i W_Q (\mathbf{u}_j W_K + \mathbf{r}_{ij}^K)$. This is identical to standard self-attention (W_Q and W_K are the query and key parameter matrices), except for the term $\mathbf{r}_{ij}^K$, which is an embedding that represents a relation between $\mathbf{u}_i$ and $\mathbf{u}_j$ from a closed set of possible relations. For example, if both tokens correspond to schema tables, an embedding will represent whether there is a primary-foreign key relation between the tables. If one of the tokens is an utterance word and another is a table column, a relation will denote if there is a string match between them. The same principle is also applied for representing the self-attention *values*, where another relation embedding matrix is used. We refer the reader to the RAT-SQL paper for details.

Overall, RAT-SQL jointly encodes the utterance, schema, the structure of the schema and alignments between the utterance and schema, and leads to state-of-the-art results in text-to-SQL parsing.

RAT-SQL layers are typically stacked on top of a pre-trained language model, such as BERT (Devlin et al., 2019). In this work, we use GRAPPA (Yu et al., 2020), a recent pre-trained model that has obtained state-of-the-art results in text-to-SQL parsing. GRAPPA is based on ROBERTA (Liu et al., 2019), but is further fine-tuned on synthetically generated utterance-query pairs using an objective for aligning the utterance and query.

Autoregressive top-down decoding The prevailing method for decoding in semantic parsing has been grammar-based autoregressive top-down decoding (Yin and Neubig, 2017; Krishnamurthy et al., 2017; Rabinovich et al., 2017), which guarantees decoding of syntactically valid programs. Specifically, the target program is represented as an abstract syntax tree under the grammar of the formal language, and linearized to a sequence of rules (or actions) using a top-down depth-first traversal. Once the program is represented as a sequence, it can be decoded using a standard sequence-to-sequence model with encoder attention (Dong and Lapata, 2016), often combined with beam search. We refer the reader to the aforementioned papers for further details on grammar-based decoding.

We now turn to describe our method, which provides a radically different approach for decoding in semantic parsing.

Algorithm 1: SMBOP

```
1 input: utterance x, schema S
2 x, s ← Encode_RAT(x, S)
3 Z_0 ← Top-K schema constants and DB values
4 for t ← 0 ... T − 1 do
5     Z'_t ← Attention(Z_t, x, x)
6     F_{t+1} ← Score-frontier(Z'_t)
7     F'_{t+1} ← arg max_K(F_{t+1})
8     Z_{t+1} ← Represent-beam(Z_t, F'_{t+1})
9 return arg max_z(Z_T)
```

3 The SMBOP parser

We first provide a high-level overview of SMBOP (see Algorithm 1 and Figure 1). As explained in §2, we encode the utterance and schema with a RAT-SQL encoder. We initialize the beam (line 3) with the K highest scoring trees of height 0, which include either schema constants or DB values. All trees are scored independently and in parallel, in a procedure formally defined in §3.3.

Next, we start the search procedure. At every step t, attention is used to contextualize the trees with information from input question representation (line 5). This representation is used to score every tree on the *frontier*: the set of sub-trees of depth $\leq t+1$ that can be constructed from sub-trees on the beam with depth $\leq t$ (lines 6-7). After choosing the top-K trees for step $t+1$, we compute a new representation for them (line 8). Finally, we return the top-scoring tree from the final decoding step, T. Steps in our model operate on tree representations independently, and thus each step is efficiently parallelized.

SMBOP resembles beam search as in each step it holds the top-K trees of a fixed height. It is also related to (pruned) chart parsing, since trees at step $t+1$ are computed from trees that were found at step t. This is unlike sequence-to-sequence models where items on the beam are competing hypotheses without any interaction.

We now provide the details of our parser. First, we describe the formal language (§3.1), then we provide precise details of our model architecture (§3.2) including beam initialization (§3.3, we describe the training procedure (§3.4), and last, we discuss the properties of SMBOP compared to prior work (§3.5).

Operation	Notation	Input → Output
Set Union	$\cup$	$R \times R \rightarrow R$
Set Intersection	$\cap$	$R \times R \rightarrow R$
Set difference	$\setminus$	$R \times R \rightarrow R$
Selection	σ	$P \times R \rightarrow R$
Cartesian product	$\times$	$R \times R \rightarrow R$
Projection	Π	$C' \times R \rightarrow R$
And	$\wedge$	$P \times P \rightarrow P$
Or	$\vee$	$P \times P \rightarrow P$
Comparison	$\{\leq, \geq, =, \neq\}$	$C \times C \rightarrow P$
Constant Union	$\sqcup$	$C' \times C' \rightarrow C'$
Order by	$\tau_{asc/dsc}$	$C \times R \rightarrow R$
Group by	γ	$C \times R \rightarrow R$
Limit	λ	$C \times R \rightarrow R$
In/Not In	$\in, \notin$	$C \times R \rightarrow P$
Like/Not Like	$\sim, \nsim$	$C \times C \rightarrow P$
Aggregation	$\mathcal{G}_{\text{agg}}$	$C \rightarrow C$
Distinct	δ	$C \rightarrow C$
Keep	κ	Any → Any

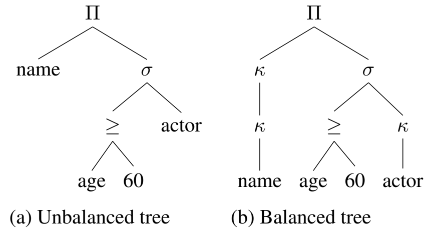
Table 1: Our relational algebra grammar, along with the input and output semantic types of each operation. P: Predicate, R: Relation, C: schema constant or DB value, C': A set of constants/values, and agg $\in$ {sum, max, min, count, avg}.

3.1 Representation of Query Trees

Relational algebra Guo et al. (2019) have shown recently that the mismatch between natural language and SQL leads to parsing difficulties. Therefore, they proposed SemQL, a formal query language with better alignment to natural language.

In this work, we follow their intuition, but instead of SemQL, we use the standard query language *relational algebra* (Codd, 1970). Relational algebra describes queries as trees, where leaves (terminals) are schema constants or DB values, and inner nodes (non-terminals) are operations (see Table 1). Similar to SemQL, its alignment with natural language is better than SQL. However, unlike SemQL, it is an existing query language, commonly used by SQL execution engines for query planning.

We write a grammar for relational algebra, augmented with SQL operators that are missing from relational algebra. We then implement a transpiler that converts SQL queries to relational algebra for parsing, and then back from relational algebra to SQL for evaluation. Table 1 shows the full grammar, including the input and output semantic types of all operations. A relation (R) is a tuple (or tuples), a predicate (P) is a Boolean condition (evaluating to `True` or `False`), a constant (C) is a schema constant or DB value, and (C') is a set of constants/values. Figure 2 shows an example relational algebra tree with the corresponding SQL query. More examples illustrating the correspondence between SQL and relational algebra (e.g., for the SQL `JOIN` operation) are in Appendix B. While our relational algebra grammar can also be adapted for standard top-down autoregressive parsing, we leave this endeavour for future work.



(a) Unbalanced tree (b) Balanced tree

Figure 2: An unbalanced and balanced relational algebra tree (with the unary KEEP operation) for the utterance *"What are the names of actors older than 60?"*, where the corresponding SQL query is `SELECT name FROM actor WHERE age ≥ 60`.

Tree balancing Conceptually, at each step SMBOP should generate new trees of height $\leq t+1$ and keep the top-K trees computed so far. In practice, it is convenient to assume that trees are balanced. Thus, we want the beam at step t to only have trees that are of height exactly t (*t-high trees*).

To achieve this, we introduce a unary KEEP operation that does not change the semantics of the subtree it is applied on. Hence, we can always grow the height of trees in the beam without changing the formal query. For training (which we elaborate on in §3.4), we balance all relational algebra trees in the training set using the KEEP operation, such that the distance from the root to all leaves is equal. For example, in Figure 2, two KEEP operations are used to balance the column `actor.name`. After tree balancing, all constants and values are at height 0, and the goal of the parser at step t is to generate the gold set of t-high trees.

3.2 Model Architecture

To fully specify Alg. 1, we need to define the following components: (a) scoring of trees on the frontier (lines 5-6), (b) representation of trees (line 8), and (c) representing and scoring of constants and DB values during beam initialization (leaves). We now describe these components. Figure 3 illustrates the scoring and representation of a binary operation.

Scoring with contextualized beams SMBOP maintains at each decoding step a beam $Z_t =$

$((z_1^{(t)}, \mathbf{z}_1^{(t)}), \dots, (z_K^{(t)}, \mathbf{z}_K^{(t)}))$, where $z_i^{(t)}$ is a symbolic representation of the query tree, and $\mathbf{z}_i^{(t)}$ is its corresponding vector representation. Unlike standard beam search, trees on our beams do not only compete with one another, but also *compose* with each other (similar to chart parsing). For example, in Fig. 1, the beam Z_0 contains the column `age` and the value `60`, which compose using the $\geq$ operator to form the `age` $\geq$ `60` tree.

We contextualize tree representations on the beam using cross-attention. Specifically, we use standard attention (Vaswani et al., 2017) to give tree representations access to the input question: $Z_t' \leftarrow \text{Attention}(Z_t, \mathbf{x}, \mathbf{x})$, where the tree representations $(\mathbf{z}_1^{(t)}, \dots, \mathbf{z}_K^{(t)})$ are the queries, and the input tokens $(\mathbf{x}_1, \dots, \mathbf{x}_{|x|})$ are the keys and values.

Next, we compute scores for all $(t+1)$-high trees on the frontier. Trees can be generated by applying either a unary (including KEEP) operation $u \in \mathcal{U}$ or binary operation $b \in \mathcal{B}$ on beam trees. Let $\mathbf{w}_u$ be a *scoring vector* for a unary operation (such as $\mathbf{w}_\kappa$, $\mathbf{w}_\delta$, etc.), let $\mathbf{w}_b$ be a *scoring vector* for a binary operation (such as $\mathbf{w}_\sigma$, $\mathbf{w}_\Pi$, etc.), and let $\mathbf{z}_i', \mathbf{z}_j'$ be contextualized tree representations on the beam. We define a scoring function for frontier trees, where the score for a new tree z_{new} generated by applying a unary rule u on a tree z_i is defined as follows:

$$s(z_{\text{new}}) = \mathbf{w}_u^\top FF_U([\mathbf{z}_i; \mathbf{z}_i']),$$

where FF_U is a 2-hidden layer feed-forward layer with relu activations, and $[\cdot;\cdot]$ denotes concatenation. Similarly the score for a tree generated by applying a binary rule b on the trees z_i, z_j is:

$$s(z_{\text{new}}) = \mathbf{w}_b^\top FF_B([\mathbf{z}_i; \mathbf{z}_i'; \mathbf{z}_j; \mathbf{z}_j']),$$

where FF_B is another 2-hidden layer feed-forward layer with relu activations.

We use semantic types to detect invalid rule applications and fix their score to $s(z_{\text{new}}) = -\infty$. This guarantees that the trees SMBOP generates are well-formed, and the resulting SQL is executable. Overall, the total number of trees on the frontier is $\leq K|\mathcal{U}| + K^2|\mathcal{B}|$. Because scores of different trees on the frontier are independent, they are efficiently computed in parallel. Note that we score new trees from the frontier *before* creating a representation for them, which we describe next.

Recursive tree representation after scoring the frontier, we generate a recursive vector representation for the top-K trees. While scoring is done with

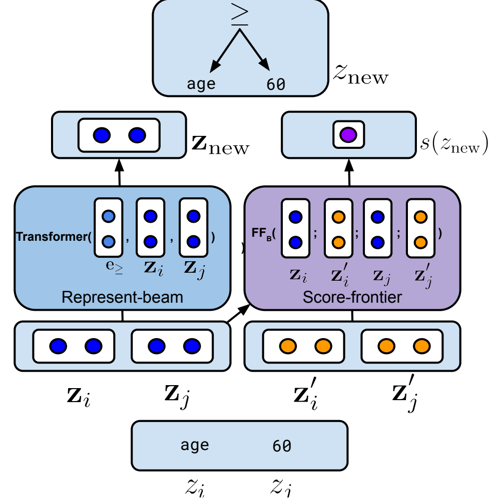


Figure 3: Illustration of our tree scoring and representation mechanisms. z is the symbolic tree, $\mathbf{z}$ is its vector representation, and $\mathbf{z}'$ its contextualized representation.

contextualized trees, representations are *not* contextualized. We empirically found that contextualized tree representations slightly reduce performance, possibly due to optimization issues.

We represent trees with another standard Transformer layer. Let $\mathbf{z}_{\text{new}}$ be the representation for a new tree, let e_ℓ be an embedding for a unary or binary operation, and let $\mathbf{z}_i, \mathbf{z}_j$ be non-contextualized tree representations from the beam we are extending. We compute a new representation as follows:

$$\mathbf{z}_{\text{new}} = \begin{cases} \text{Transformer}(\mathbf{e}_\ell, \mathbf{z}_i) & \text{unary } \ell \\ \text{Transformer}(\mathbf{e}_\ell, \mathbf{z}_i, \mathbf{z}_j) & \text{binary } \ell \\ \mathbf{z}_i & \ell = \text{KEEP} \end{cases}$$

where for the unary KEEP operation, we simply copy the representation from the previous step.

Return value As mentioned, the parser returns the highest-scoring tree in Z_T. More precisely, we return the highest-scoring *returnable* tree, where a returnable tree is a tree that has a valid semantic type, that is, Relation (R).

3.3 Beam initialization

As described in Line 3 of Alg. 1, the beam Z_0 is initialized with K schema constants (e.g., `actor`, `age`) and DB values (e.g., `60`, `"France"`). In particular, we independently score schema constants and choose the top-$\frac{K}{2}$, and similarly score DB values and choose the top-$\frac{K}{2}$, resulting in a total beam of size K.

Schema constants We use a simple scoring function $f_{\text{const}}(\cdot)$. Recall that $\mathbf{s}_i$ is a representation of a

constant, contextualized by the rest of the schema and the utterance. The function $f_{\text{const}}(\cdot)$ is a feed-forward network that scores each schema constant independently: $f_{\text{const}}(\mathbf{s}_i) = \mathbf{w}_{\text{const}} \tanh(W_{\text{const}}\mathbf{s}_i)$, and the top-$\frac{K}{2}$ constants are placed in Z_0.

DB values Because the number of values in the DB is potentially huge, we do not score all DB values. Instead, we learn to detect spans in the question that correspond to DB values. This leads to a setup that is similar to extractive question answering (Rajpurkar et al., 2016), where the model outputs a distribution over input spans, and thus we adopt the architecture commonly used in extractive question answering. Concretely, we compute the probability that a token is the start token of a DB value, $P_{\text{start}}(x_i) \propto \exp(\mathbf{w}_{\text{start}}^\top \mathbf{x}_i)$, and similarly the probability that a token is the end token of a DB value, $P_{\text{end}}(x_i) \propto \exp(\mathbf{w}_{\text{end}}^\top \mathbf{x}_i)$, where $\mathbf{w}_{\text{start}}$ and $\mathbf{w}_{\text{end}}$ are parameter vectors. We define the probability of a span $(x_i, \ldots, x_j)$ to be $P_{\text{start}}(x_i) \cdot P_{\text{end}}(x_j)$, and place in the beam Z_0 the top-$\frac{K}{2}$ input spans, where the representation of a span (x_i, x_j) is the average of $\mathbf{x}_i$ and $\mathbf{x}_j$.

A current limitation of SMBOP is that it cannot generate DB values that do not appear in the input question. This would require adding a mechanism such as "BRIDGE" proposed by Lin et al. (2020).

3.4 Training

To specify the loss function, we need to define the supervision signal. Recall that given the gold SQL program, we convert it into a gold *balanced relational algebra tree* z^{gold}, as explained in §3.1 and Figure 2. This lets us define for every decoding step the set of t-high gold sub-trees $\mathcal{Z}_t^{\text{gold}}$. For example $\mathcal{Z}_0^{\text{gold}}$ includes all gold schema constants and input spans that match a gold DB value,[3] $\mathcal{Z}_1^{\text{gold}}$ includes all 1-high gold trees, etc.

During training, we apply "bottom-up Teacher Forcing" (Williams and Zipser, 1989), that is, we populate[4] the beam Z_t with all trees from $\mathcal{Z}_t^{\text{gold}}$ and then fill the rest of the beam (of size K) with the top-scoring non-gold predicted trees. This guarantees that we will be able to compute a loss at each decoding step, as described below.

Loss function During search, our goal is to give high scores to the possibly multiple *sub-trees* of

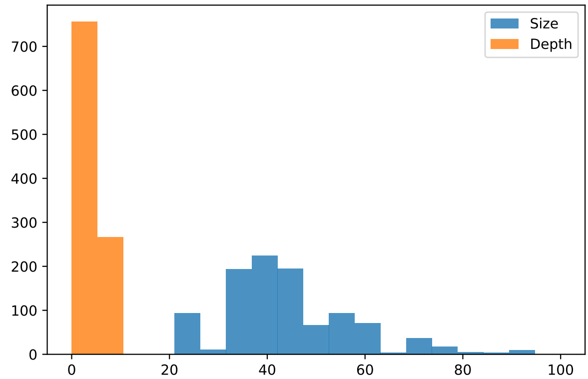


Figure 4: A histogram showing the distribution of the height of relational algebra trees in SPIDER, and the size of equivalent SQL query trees.

the gold tree. Because of teacher forcing, the frontier F_{t+1} is guaranteed to contain all gold trees $\mathcal{Z}_{t+1}^{\text{gold}}$. We first apply a softmax over all frontier trees $p(z_{\text{new}}) = \text{softmax}\{s(z_{\text{new}})\}_{z_{\text{new}} \in F_{t+1}}$ and then maximize the probabilities of gold trees:

$$\frac{1}{C} \sum_{t=0}^{T} \sum_{z_t \in \mathcal{Z}_t^{\text{gold}}} \log p(z_t)$$

where the loss is normalized by C, the total number of summed terms. In the initial beam, Z_0, the probability of an input span is the product of the start and end probabilities, as explained in §3.3.

3.5 Discussion

To our knowledge, this work is the first to present a semi-autoregressive bottom-up semantic parser. We discuss the benefits of our approach.

SMBOP has theoretical runtime complexity that is logarithmic in the size of the tree instead of linear for autoregressive models. Figure 4 shows the distribution over the height of relational algebra trees in SPIDER, and the size of equivalent SQL query trees. Clearly, the height of most trees is at most 10, while the size is 30-50, illustrating the potential of this approach. In §4, we demonstrate that indeed semi-autoregressive parsing leads to substantial empirical speed-up.

Unlike top-down autoregressive models, SMBOP naturally computes representations $\mathbf{z}$ for all sub-trees constructed at decoding time, which are well-defined semantic objects. These representations can be used in setups such as *contextual semantic parsing*, where a semantic parser answers a sequence of questions. For example, given the

[3] In Spider, in 98.2% of the training examples, all gold DB values appear as input spans.

[4] We compute this through an efficient tree hashing procedure. See Appendix A.

questions *"How many students are living in the dorms?"* and then *"what are their last names?"*, the pronoun *"their"* refers to a sub-tree from the SQL tree of the first question. Having a representation for such sub-trees can be useful when parsing the second question, in benchmarks such as SPARC (Yu et al., 2019).

Another potential benefit of bottom-up parsing is that sub-queries can be executed while parsing (Berant et al., 2013; Liang et al., 2017), which can guide the search procedure. Recently, Odena et al. (2020) proposed such an approach for program synthesis, and showed that conditioning on the results of execution can improve performance. We do not explore this advantage of bottom-up parsing in this work, since executing queries at training time leads to a slow-down during training.

SMBOP is a bottom-up semi-autoregressive parser, but it could potentially be modified to be autoregressive by decoding one tree at a time. Past work (Cheng et al., 2019) has shown that the performance of bottom-up and top-down autoregressive parsers is similar, but it is possible to re-examine this given recent advances in neural architectures.

4 Experimental Evaluation

We conduct our experimental evaluation on SPIDER (Yu et al., 2018), a challenging large-scale dataset for text-to-SQL parsing. SPIDER has become a common benchmark for evaluating semantic parsers because it includes complex SQL queries and a realistic zero-shot setup, where schemas at test time are different from training time.

4.1 Experimental setup

We encode the input utterance x and the schema S with GRAPPA, consisting of 24 Transformer layers, followed by another 8 RAT-SQL layers, which we implement inside AllenNLP (Gardner et al., 2018). Our beam size is $K = 30$, and the number of decoding steps is $T = 9$ at inference time, which is the maximal tree depth on the development set. The transformer used for tree representations has one layer, 8 heads, and dimensionality 256. We train for 60K steps with batch size 60, and perform early stopping based on the development set.

Evaluation We evaluate performance with the official SPIDER evaluation script, which computes *exact match (EM)*, i.e., whether the predicted SQL query is identical to the gold query after some query normalization. The evaluation script uses *anonymized* queries, where DB values are converted to a special `value` token. In addition, for models that output DB values, the evaluation script computes *denotation accuracy*, that is, whether executing the output SQL query results in the right denotation (answer). As SMBOP generates DB values, we evaluate using both EM and denotation accuracy

Model	EM	Exec
RAT-SQL+GP+GRAPPA	**69.8**%	n/a
RAT-SQL+GAP	69.7%	n/a
RAT-SQL+GRAPPA	69.6%	n/a
RAT-SQL+STRUG	68.4%	n/a
BRIDGE+BERT (ensemble)	67.5%	68.3
RAT-SQLv3+BERT	65.6%	n/a
SMBOP+GRAPPA	69.5%	**71.1**%

Table 2: Results on the SPIDER test set.

Models We compare SMBOP to the best non-anonymous models on the SPIDER leaderboard at the time of writing. Our model is most comparable to RAT-SQL+GRAPPA, which has the same encoder, but an autoregressive decoder.

In addition, we perform the following ablations and oracle experiments:

- NO X-ATTENTION: We remove the cross attention that computes Z'_t and uses the representations in Z_t directly to score the frontier. In this setup, the decoder only observes the input question through the 0-high trees in Z_0.
- WITH CNTX REP.: We use the contextualized representations not only for *scoring*, but also as input for creating the new trees Z_{t+1}. This tests if contextualized representations on the beam hurt or improve performance.
- NO DB VALUES: We anonymize all SQL queries by replacing DB values with `value`, as described above, and evaluate SMBOP using EM. This tests whether learning from DB values improves performance.
- Z_0-ORACLE: An oracle experiment where Z_0 is populated with the gold schema constants (but predicted DB values). This shows results given perfect schema matching.

4.2 Results

Table 2 shows test results of SMBOP compared to the top (non-anonymous) entries on the leaderboard (Zhao et al., 2021; Shi et al., 2021; Yu et al., 2020; Deng et al., 2020; Lin et al., 2020; Wang et al., 2020a). SMBOP obtains an EM of 69.5%, only

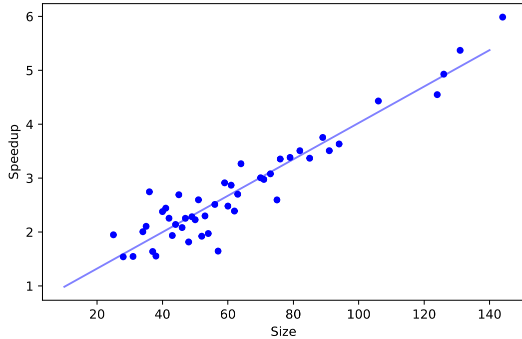


Figure 5: Speed-up on the development set compared to autoregressive decoding, w.r.t the size of the SQL query.

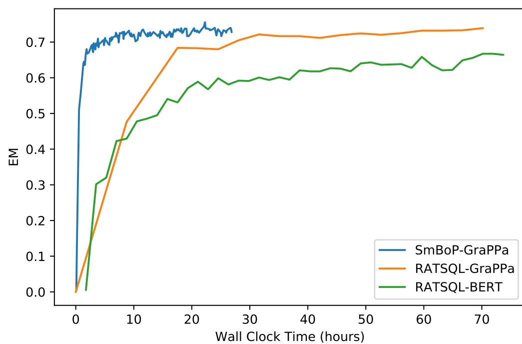


Figure 6: EM as a function of wall clock time on the development set of SPIDER during training.

0.3% lower than the best model, and 0.1% lower than RAT-SQL+GRAPPA, which has the same encoder, but an autoregressive decoder. Moreover, SMBOP outputs DB values, unlike other models that output anonymized queries that cannot be executed. SMBOP establishes a new state-of-the-art in denotation accuracy, surpassing an ensemble of BRIDGE+BERT models by 2.9 denotation accuracy points, and 2 EM points.

Turning to decoding time, we compare SMBOP to RAT-SQLv3+BERT, since the code for RAT-SQLv3+GRAPPA was not available. To the best of our knowledge the decoder in both is identical, so this should not affect decoding time. We find that the decoder of SMBOP is on average 2.23x faster than the autoregressive decoder on the development set. Figure 5 shows the average speed-up for different query tree sizes, where we observe a clear linear speed-up as a function of query size. For long queries the speed-up factor reaches 4x-6x. When including also the encoder, the average speed-up obtained by SMBOP is 1.55x.

In terms of training time, SMBOP leads to

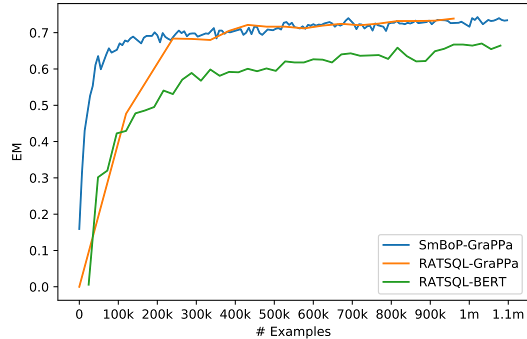


Figure 7: EM as a function of the number of examples on the development set of SPIDER during training.

Model	Exec	EM	BEM	Z_0 REC.
RAT-SQL+GRAPPA	n/a	73.9%	n/a	n/a
SMBOP	75.0%	74.7%	82.6%	98.3%
- NO X-ATT	74.8%	72.7%	81.1%	96.6%
- WITH CNTX REP.	73.4%	72.4%	81.9%	97.3%
- NO DB VALUES	n/a	71.3%	80.0%	98.1%
SMBOP-Z_0-ORACLE	77.4%	79.1%	85.8%	n/a

Table 3: Development set EM, beam EM (BEM) and recall on schema constants and DB values (Z_0 rec.) for all models.

much faster training and convergence. We compare the learning curves of SMBOP and RAT-SQLv3+BERT, both trained on an RTX 3090, and also to RAT-SQLv3+GRAPPA using performance as a function of the number of examples, sent to us in a personal communication from the authors. SMBOP converges much faster than RAT-SQL (Fig. 7). E.g., after 120K examples, the EM of SMBOP is 67.5, while for RAT-SQL+GRAPPA it is 47.6. Moreover, SMBOP processes at training time 20.4 examples per second, compared to only 3.8 for the official RAT-SQL implementation. Combining these two facts leads to much faster training time (Fig. 6), slighly more than one day for SMBOP vs. 5-6 days for RAT-SQL.

Ablations Table 3 shows results of ablations on the development set. Apart from EM, we also report: (a) beam EM (BEM): whether a correct tree was found *anywhere* during the T decoding steps, and (b) Z_0 *recall*: the fraction of examples where the parser placed all gold schema constants and DB values in Z_0. This estimates the ability of our models to perform schema matching in a single non-autoregressive step.

We observe that ablating cross-attention leads to a small reduction in EM. This rather small drop is surprising since it means that all information about the question is passed to the decoder through

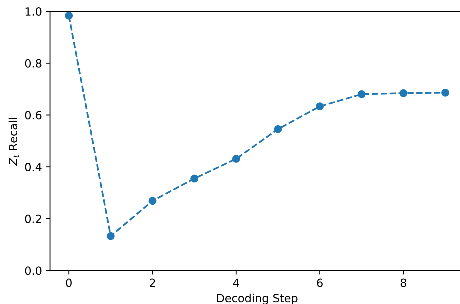


Figure 8: Z_t Recall across decoding steps.

Z_0. We hypothesize that this is possible, because the number of decoding steps is small, and thus utterance information can propagate through the decoder. Using contextualized representations for trees also leads to a small drop in performance. Last, we see that feeding the model with actual DB values rather than an anonymized `value` token improves performance by 3.4 EM points.

Looking at Z_0 RECALL, we see that models perform well at detecting relevant schema constants and DB values (96.6%-98.3%), despite the fact that this step is fully non-autoregressive. However, an oracle model that places all gold schema constants and only gold schema constants in Z_0 further improves EM (74.7 →79.1%), with a BEM of 85.8%. This shows that better schema matching and search can still improve performance on SPIDER.

BEM is 8%-9% higher than EM, showing that, similar to past findings in semantic parsing (Goldman et al., 2018; Yin and Neubig, 2019), adding a re-ranker on top of the trees computed by SMBOP can potentially improve performance. We leave this for future work.

4.3 Analysis

We extend the notion of Z_0 recall to all decoding steps, where Z_t recall is whether all gold t-high sub-trees were generated at step t. We see Z_t recall across decoding steps in Figure 8.[5] The drop after step 0 and subsequent rise indicate that the model maintains in the beam, using the KEEP operation, trees that are sub-trees of the gold tree, and expands them in later steps. This means that the parser can recover from errors in early decoding steps as long as the relevant trees are kept on the beam.

To better understand search errors we perform the following analysis. For each example, we find the first gold tree that is dropped from the beam (if there is more than one, we choose one randomly). We then look at the children of t, and see whether at least one was expanded in some later step in decoding, or whether the children were completely abandoned by the search procedure. We find that in 62% of the cases indeed one of the children was incorrectly expanded, indicating a composition error.

In this work, we used beam size $K = 30$. Reducing K to 20 leads to a drop of less than point (74.7→73.8), and increasing K to 40 reduces performance by (74.7→72.6). In all cases, decoding time does not dramatically change.

Last, we randomly sample 50 errors from SMBOP and categorize them into the following types:

- Search errors (52%): we find that most search errors are due to either extra or missing `JOIN` or `WHERE` conditions .
- Schema encoding errors (34%): Missing or extra schema constants in the predicted query.
- Equivalent queries (12%): Predicted trees that are equivalent to the gold tree, but the automatic evaluation script does not handle.

5 Conclusions

In this work we present the first semi-autoregressive bottom-up semantic parser that enjoys logarithmic theoretical runtime, and show that it leads to a 2.2x speed-up in decoding and ∼5x faster taining, while maintaining state-of-the-art performance. Our work shows that bottom-up parsing, where the model learns representations for semantically meaningful sub-trees is a promising research direction, that can contribute in the future to setups such as contextual semantic parsing, where sub-trees often repeat, and can enjoy the benefits of execution at training time. Future work can also leverage work on learning tree representations (Shiv and Quirk, 2019) to further improve parser performance.

Acknowledgments

We thank Tao Yu, Ben Bogin, Jonathan Herzig, Inbar Oren, Elad Segal and Ankit Gupta for their useful comments. This research was partially supported by The Yandex Initiative for Machine Learning, and the European Research Council (ERC) under the European Union Horizons 2020 research and innovation programme (grant ERC DELPHI 802800).

[5]This metric checks for exact sub-tree match, unlike EM that does more normalization, so numbers are not comparable to EM.

References

J. Berant, A. Chou, R. Frostig, and P. Liang. 2013. Semantic parsing on Freebase from question-answer pairs. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Jianpeng Cheng, Siva Reddy, Vijay Saraswat, and Mirella Lapata. 2019. Learning an executable neural semantic parser. *Computational Linguistics*, 45(1):59–94.

James Clarke, Dan Goldwasser, Ming-Wei Chang, and Dan Roth. Driving semantic parsing from the world's response. In *Proceedings of the Fourteenth Conference on Computational Natural Language Learning (CoNLL)*.

E. F. Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387.

Xiang Deng, Ahmed Hassan Awadallah, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. 2020. Structure-grounded pretraining for text-to-sql. *arXiv preprint arXiv:2010.12773*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 33–43, Berlin, Germany. Association for Computational Linguistics.

Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew Peters, Michael Schmitz, and Luke Zettlemoyer. 2018. AllenNLP: A deep semantic natural language processing platform. In *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*, pages 1–6, Melbourne, Australia. Association for Computational Linguistics.

Omer Goldman, Veronica Latcinnik, Ehud Nave, Amir Globerson, and Jonathan Berant. 2018. Weakly supervised semantic parsing with abstract examples. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL) (Volume 1: Long Papers)*, pages 1809–1819, Melbourne, Australia. Association for Computational Linguistics.

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 *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 4524–4535, Florence, Italy. Association for Computational Linguistics.

Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Eisenschlos. 2020. TaPas: Weakly supervised table parsing via pre-training. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4320–4333, Online. Association for Computational Linguistics.

Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. 2017. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

C. Liang, J. Berant, Q. Le, and K. D. F. N. Lao. 2017. Neural symbolic machines: Learning semantic parsers on Freebase with weak supervision. In *Association for Computational Linguistics (ACL)*.

Percy Liang, Michael Jordan, and Dan Klein. 2011. Learning dependency-based compositional semantics. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-HLT)*, pages 590–599, Portland, Oregon, USA. Association for Computational Linguistics.

Xi Victoria Lin, Richard Socher, and Caiming Xiong. 2020. Bridging textual and tabular data for cross-domain text-to-SQL semantic parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4870–4888, Online. Association for Computational Linguistics.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Ralph C. Merkle. 1987. A digital signature based on a conventional encryption function. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, volume 293 of *Lecture Notes in Computer Science*.

Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, and Charles Sutton. 2020. Bustle: Bottom-up program-synthesis through learning-guided exploration.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1139–1149, Vancouver, Canada. Association for Computational Linguistics.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas. Association for Computational Linguistics.

Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. 2020. Green AI. *Communications of the ACM*, 63.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), Volume 2 (Short Papers)*, pages 464–468, New Orleans, Louisiana. Association for Computational Linguistics.

Peng Shi, Patrick Ng, Zhiguo Wang, Henghui Zhu, Alexander Hanbo Li, Jun Wang, Cicero Nogueira dos Santos, and Bing Xiang. 2021. Learning contextual representations for semantic parsing with generation-augmented pre-training. *arXiv preprint arXiv:2012.10309*.

Vighnesh Leonardo Shiv and Chris Quirk. 2019. Novel positional encodings to enable tree-structured transformers. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NeurIPS)*.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020a. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020b. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*.

Ronald J. Williams and David Zipser. 1989. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics.

Pengcheng Yin and Graham Neubig. 2019. Reranking for neural semantic parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for joint understanding of textual and tabular data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8413–8426, Online. Association for Computational Linguistics.

Tao Yu, Chien-Sheng Wu, Xi Victoria Lin, Bailin Wang, Yi Chern Tan, Xinyi Yang, Dragomir Radev, Richard Socher, and Caiming Xiong. 2020. Grappa: Grammar-augmented pre-training for table semantic parsing. *arXiv preprint arXiv:2009.13845*.

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 (EMNLP)*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Irene Li Heyang Er, Bo Pang, Tao Chen, Emily Ji, Shreya Dixit, David Proctor, Sungrok Shim, Vincent Zhang Jonathan Kraft, Caiming Xiong, Richard Socher, and Dragomir Radev. 2019. Sparc: Cross-domain semantic parsing in context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, Florence, Italy. Association for Computational Linguistics.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI)*.

Luke S. Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence (UAI)*.

Liang Zhao, Hexin Cao, and Yunsong Zhao. 2021. Gp: Context-free grammar pre-training for text-to-sql parsers.

A Computing supervision through tree hashing

In every decoding step t, we wish to compute for every tree z_{new} in the frontier F_{t+1} if $z_{\text{new}} \in \mathcal{Z}_t^{\text{gold}}$. This is achieved using tree hashing. First, during preprocessing, for every height t, we compute the gold hashes h_t^{gold}, the hash values of every sub-tree of z^{gold} of height t, in a recursive fashion using a Merkle tree hash (Merkle, 1987). Specifically, we define:

$$\text{hash}(z) = g(\text{label}(z), \text{hash}(z_l), \text{hash}(z_r))$$

Where g is a simple hash function, z_l, z_r are the left and right children of z, and $\text{label}(\cdot)$ gives the node type (such as σ and Π).

During training, in each decoding step t, since the hash function is defined recursively, we can compute the frontier hashes using the hash values of the current beam. Then, for every frontier hash we can perform a lookup to check if $\text{hash}(z) \in h_t^{\text{gold}}$. Both the hash computation and lookup are done in parallel for all frontier trees using the GPU.

B Examples for Relational Algebra Trees

We show multiple examples of relation algebra trees along with the corresponding SQL query, for better understanding of the mapping between the two.

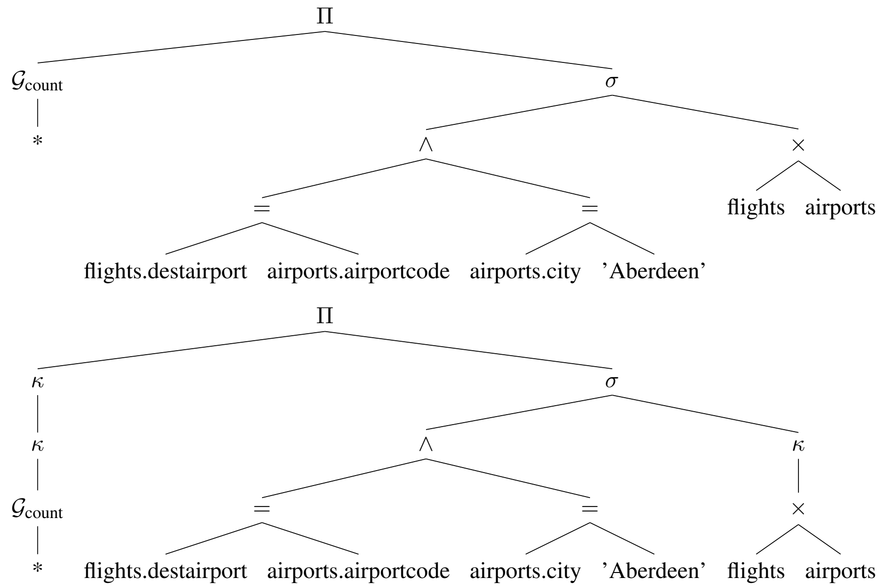


Figure 9: Unbalanced and balanced relational algebra trees for the utterance *"How many flights arriving in Aberdeen city?"*, where the corresponding SQL query is `SELECT COUNT( * ) FROM flights JOIN airports ON flights.destairport = airports.airportcode WHERE airports.city = 'Aberdeen'`.

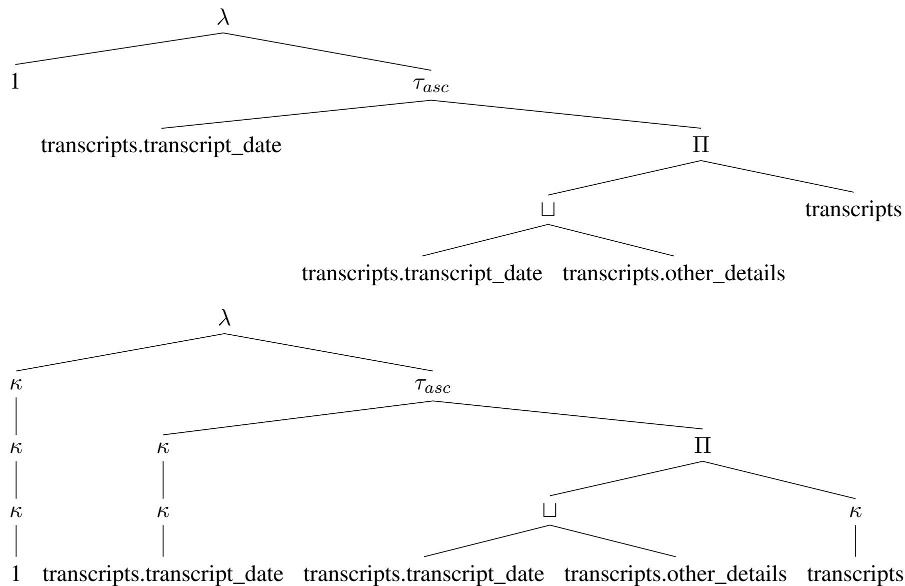


Figure 10: Unbalanced and balanced relational algebra trees for the utterance *"When is the first transcript released? List the date and details."*, where the corresponding SQL query is `SELECT transcripts.transcript_date , transcripts.other_details FROM transcripts ORDER BY transcripts.transcript_date ASC LIMIT 1`.

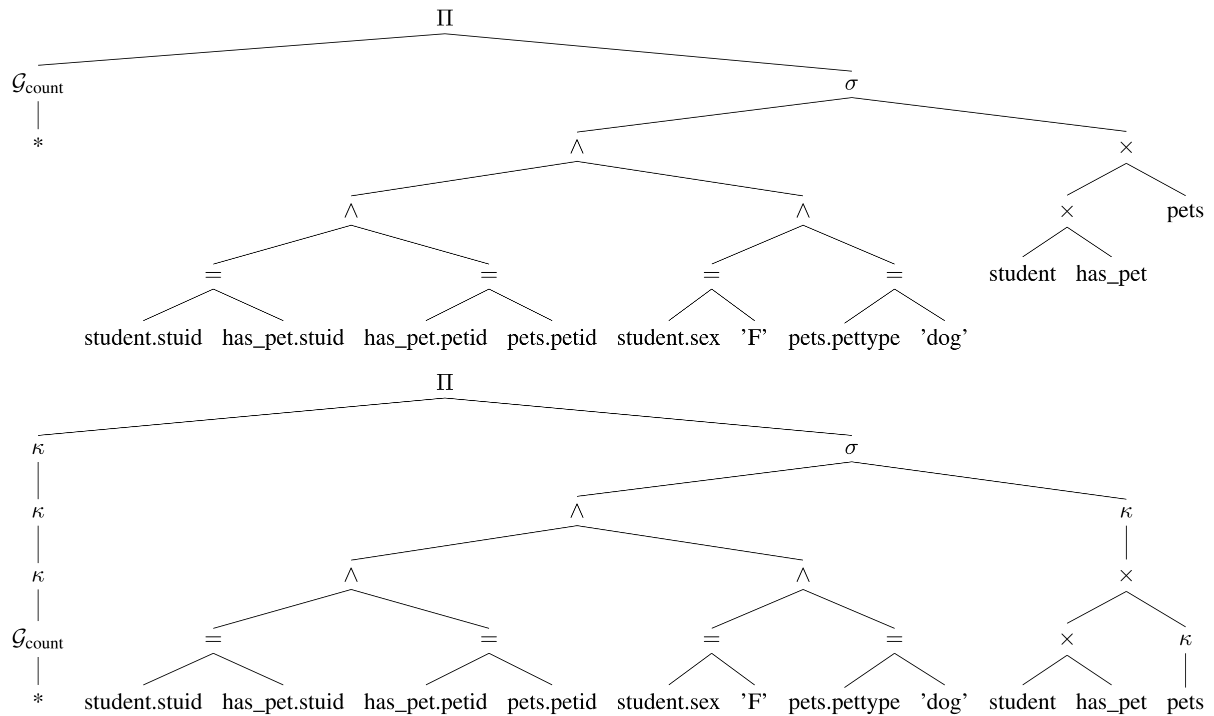


Figure 11: Unbalanced and balanced relational algebra trees for the utterance *"How many dog pets are raised by female students?"*, where the corresponding SQL query is `SELECT COUNT( * ) FROM student JOIN has_pet ON student.stuid = has_pet.stuid JOIN pets ON has_pet.petid = pets.petid WHERE student.sex = 'F' AND pets.pettype = 'dog'`.

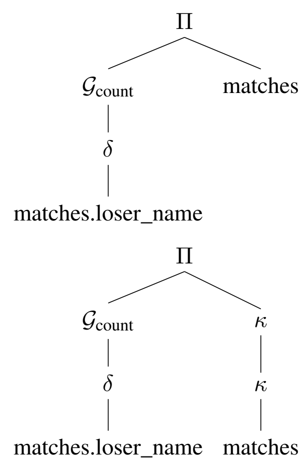


Figure 12: Unbalanced and balanced relational algebra trees for the utterance *"Find the number of distinct name of losers."*, where the corresponding SQL query is `SELECT COUNT( DISTINCT matches.loser_name ) FROM matches`.