Multitask Pretraining with Structured Knowledge for Text-to-SQL Generation

Robert Giaquinto, Dejiao Zhang, Benjamin Kleiner, Yang Li
Ming Tan, Parminder Bhatia, Ramesh Nallapati, Xiaofei Ma

AWS AI Labs
{rgiaq,dejiaoz,kleinerb,ylizam,
mingtan,parmib,rnallapa,xiaofeim}@amazon.com

Abstract

Many machine learning-based low-code or no-code applications involve generating code that interacts with structured knowledge. For example, one of the most studied tasks in this area is generating SQL code from a natural language statement. Prior work shows that incorporating context information from the database schema, such as table and column names, is beneficial to model performance on this task. In this work we present a large pretraining dataset and strategy for learning representations of text, tables, and SQL code that leverages the entire context of the problem. Specifically, we build on existing encoder-decoder architecture by introducing a multitask pretraining framework that complements the unique attributes of our diverse pretraining data. Our work represents the first study on large-scale pretraining of encoder-decoder models for interacting with structured knowledge, and offers a new state-of-the-art foundation model in text-to-SQL generation.

We validate our approach with experiments on two SQL tasks, showing improvement over existing methods, including a 1.7 and 2.2 percentage point improvement over prior state-of-the-arts on Spider and CoSQL.

1 Introduction

Tables, relational databases, and other forms of structured knowledge (SK) encompass a massive amount of data across a wide range of applications. Extracting insights held in such data often requires proficiency in query languages like SQL, making it only accessible to the minority of people with the technical skills. A natural language interface, however, would expand access to these information exponentially. Likewise, querying via natural language allows users quickly hone in on an answer to their particular question, rather than visually scanning dense tables where the majority of the information is irrelevant to the user. To that end, we explore pretraining techniques for large language models that focus on the challenging interplay between structured and unstructured knowledge, and target a variety of downstream text-to-SQL tasks.

Recently there have been significant advancements in learning representations for tables (Yin et al., 2020; Herzig et al., 2020; Eisenschlos et al., 2020; Liu et al., 2020; Liu et al., 2022; Wang et al., 2021c; Yu et al., 2021; Cheng et al., 2022; Dong et al., 2022), which advanced the state-of-the-art in a range of table-to-text tasks, like table question-answering (Nan et al., 2022; Chen et al., 2021), fact verification (Chen et al., 2020; Aly et al., 2021), data-to-text (Parikh et al., 2020; Nan et al., 2021), and semantic parsing (Yu et al., 2019b; Zhong et al., 2017). While better table understanding benefits a range of tasks, pretraining focused on text-to-SQL has thus far received less attention. Pretrained encoders, such as TaBERT and TAPAS (Yu et al., 2021; Yin et al., 2020; Herzig et al., 2020), show that pretraining BERT-style encoders (Devlin et al., 2019) on tables with mask language modeling (MLM) loss produces a strong foundation model that can be extended for text-to-SQL. GRAPPA includes small amount of synthetic SQL code in the pretraining data to more specifically target the text-to-SQL task (Yu et al., 2021). These encoder-only approaches are, however, restricted in their generative capabilities as they must be combined with an additional module that is carefully designed to generate valid SQL code (Zhong et al., 2017; Wang et al., 2021a).

Encoder-decoder architectures like T5 (Raffel et al., 2020), on the other hand, exhibit better performance on text-to-SQL to-date when constraining the decoder with rules that check for syntactic correctness (Scholak et al., 2021). However, the T5-based models with exceptional text-to-SQL performance (Xie et al., 2022; Scholak et al., 2021) have still only been pretrained on natural language (NL) — begging the question, can text-to-SQL encoder-decoders benefit from pretraining on structured in-
formation or code? Most recently, Andrejczuk et al. (2022) proposed a multi-task tabular pretraining strategy for T5 model, but their work introduced the tabular knowledge to the model with a single data source, i.e. Wikipedia tables.

In this work we introduce our SQL and Table Aligned Multi-task Pretraining (STAMP) framework, which explores pretraining encoder-decoder models for text-to-SQL. Starting from text-only T5 (Raffel et al., 2020) checkpoints, our multi-stage pretraining framework refines previous text-only models by continuing training on a collection of large multi-modal datasets that combine structured knowledge with natural language and SQL. Additionally, inspired by the impressive generalization of large language models incorporating code in pretraining data (Athiwaratkun et al., 2022; Brown et al., 2020; Chowdhery et al., 2022; Du et al., 2022; Thoppilan et al., 2022), we apply our pretraining framework to CodeT5 (Wang et al., 2021b) checkpoints that are trained on code.

Building on recent work in multi-task pretraining (Tay et al., 2022; Aghajanyan et al., 2021; Sanh et al., 2022; Aribandi et al., 2021), we combine masked language modeling (MLM) with task-aware context-to-output objectives that vary across tasks and datasets. For pretraining datasets with multiple modalities (i.e. combinations of NL, SQL, and structured knowledge) or intrinsic splits (e.g. question and answer), we explore the benefit of the dual learning objectives (Wang et al., 2021b). We assess our pretraining strategy on a variety of SQL benchmarks following the UnifiedSKG framework (Xie et al., 2022). Our approach outperforms previous text- and code-only pretraining, and gives a new state-of-the-art on a range of benchmarks. To better understand our strategy, we present ablation studies on the optimal objective mix, the impact of linearizing structured knowledge into row- versus column-centric tables, and the effect of building on previously pretrained text- versus code-only checkpoints. Our work shows that continued pretraining with multi-task learning is a promising direction for advancing the capacity of language models.
2 Related Work

Encoder-only Encoder-only transformer architectures like BERT and its successors (Devlin et al., 2019; Liu et al., 2019; Joshi et al., 2020; Reimers and Gurevych, 2019; Clark et al., 2020) optimize masked language modeling (MLM) objectives while using a bidirectional receptive field covering the whole input sequence. The encoder-only architectures perform well across a variety of tasks like classification, regression, sentiment analysis, question-answering, and retrieval. However, recent work (Herzig et al., 2020; Yin et al., 2020; Yu et al., 2021) shows that tasks like table-to-text and text-to-SQL require additional pretraining on structured knowledge for good generalization, and adapting MLM objectives to the unique structure of tabular data improves learning.

Prior to BERT, text-to-SQL models like SQL-Net and Seq2SQL (Zhong et al., 2017; Xu et al., 2017) encoded inputs with bidirectional LSTMs (Hochreiter and Schmidhuber, 1997) and generated queries via slot-filling. Text-to-SQL performance improved with the adoption of BERT-based encoders, for example (Yu et al., 2021; Wang et al., 2021a) attach feed forward networks and LSTMs to the BERT-style encoder to generate queries. Because encoder-only architectures are restricted in their ability to generate sequences, they require careful design to generate valid SQL queries and limit the complexity of those queries.

Encoder-Decoder Alternatively, encoder-decoders like BART (Lewis et al., 2019) and T5 (Raffel et al., 2020) combine the bidirectional encoder with a causal decoder are naturally suited for sequence-to-sequence tasks like text-to-SQL, and are quickly becoming the mainstream approach due to the reduced need for domain specific solutions (Qin et al., 2022). T5 (Raffel et al., 2020) in particular achieves impressive performance on a range of table-to-text and text-to-SQL tasks (Xie et al., 2022) despite pretraining that is limited to NL. Moreover, Shi et al. (2020) and Liu et al. (2022) leverage a BART-style encoder-decoder to improve the performance of pretrained models for text-to-SQL and table-to-text tasks, respectively. We follow this line, proposing a strategy that builds on top of T5 and CodeT5 (Wang et al., 2021b).

Multi-Task Training Raffel et al. (2020) explore various self-supervised objectives, and found the fill-in-the-blank style of denoising objective as most effective. Additionally, combining MLM objectives with small amounts of auxiliary objectives is effective (Liu et al., 2019; Aroca-Ouellette and Rudzicz, 2020). For encoder-decoder models Tay et al. (2022); Wang et al. (2021b) show the benefit of multi-task pretraining on a mix of the T5 span corruption objective (Raffel et al., 2020) along with a the causal language modeling (CLM) style of objective, similar to those used in decoder-only architectures (Brown et al., 2020). In the domain of text-to-SQL, Yu et al. (2021); Tao Yu et al. (2021) perform multitask learning by combining MLM with SQL specific objectives. Lastly, Xie et al. (2022); Aghajanyan et al. (2021); Aribandi et al. (2021); Sanh et al. (2022); FitzGerald et al. (2022); Chen et al. (2022) demonstrate that multi-task learning across a variety of datasets can improve performance relative to the single-task, single-dataset paradigm. Wang et al. (2021b) show that an objective mix specific to programming languages (PL) along with dual learning on bimodal data promotes generation on tasks combining PL and NL.

3 Multi-Task Pretraining on Structured Knowledge

Our SQL and Table Aligned Multi-task Pretraining (STAMP) model builds on the T5 encoder-decoder architecture and pretraining checkpoints (Raffel et al., 2020), and similarly our CodeSTAMP models build on the CodeT5 architecture and checkpoints Wang et al. (2021b). We develop a multi-task pretraining framework specifically designed to leverage our large and unique collection of data that combine various data modalities, namely natural language (NL), structured knowledge (SK), and SQL. STAMP introduces a new stage of pretraining that transitions T5 from being a purely NL programing language (PL) trained model to a backbone model that excels at text-to-SQL generation.

Next, we present the construction of our pretraining dataset in Section 3.1, the mixture of objectives designed to learn the unique structure of our data and align the NL, SK, SQL data modalities in Section 3.2, and our unified format for representing tasks and structured knowledge in Section 3.3.

3.1 Datasets and Pre-Processing

Our pretraining dataset consists of 18 million examples, with various combinations of NL, SQL code, and structured knowledge (see Figure 2). Our data is derived from diverse sources and we propose dif-
Different strategies to remove many low-quality and noisy data from each data source. We tokenize the raw data using the corresponding T5 and CodeT5 tokenizers, which we augment to support new special tokens for representing input data modality, output tasks, and table structures. We process all data into sequences of up to 1024 tokens. More details on pre-processing are in Appendix A.

**Table Data** Approximately half of our pretraining data \( (N = 10,136,268) \) combine tables with NL. These table datasets derive from Wikipedia, WDC’s Web Table Corpora, and arXiv. Pretraining on table datasets acts as a bridge from the previous text-only pretraining, while promoting alignment between NL and structured knowledge. In initial experiments we pretrained on all available table and NL pairs. However, after closer examination we discovered that a significant portion these examples exhibited minimal connection between the table and NL — and hence are unlikely to promote the desired alignment. Therefore, we choose to focus on high-quality examples and remove approximately 75% of the examples in which there is a tenuous or no connection between the table and the paired NL. To identify noisy examples we compute an edit similarity between the NL and the content of the table, we then drop examples with such similarity below a threshold. Likewise, to reduce noise within each example we truncate tables, keeping at most 6 rows and 25 columns which have the highest edit similarity between table and NL.

**SQL Data** The remainder of our pretraining data incorporate SQL. Approximately 10% of the examples \( (N = 1,918,468) \) are SQL code from GitHub repositories with permissive licenses. SQL code from GitHub only includes only a small amount NL in code comments, and some structured knowledge in the database schema definitions. We filter these data to remove duplicates and repetitive statements.

Approximately 25% of the examples \( (N = 4,479,767) \) are from SQL-related posts on Stack Overflow. These data combine NL questions and answers with snippets of SQL code, thereby bridging the NL knowledge learned during the prior text-only pretraining into domain-specific language, and aligning SQL with NL. We perform augmentations to increase the number of question-answer pairs and leverage hidden human supervision \(^1\) in the data. We first create five augmented versions of each question using random word deletion, random word appending, synonym replacement, and paraphrasing. We then create up to six versions of each original example by pairing combinations of answers with augmented versions of the questions.

Lastly, approximately 11% of the examples \( (N = 2,005,456) \) in our data derive from TAPEx (Liu et al., 2022), a dataset consisting of SQL generated from templates along with their corresponding execution result. To improve the quality and better align these data with downstream tasks we perform the following modifications. First, we remove 2.3 million duplicates (of the original 5 million examples), add a FROM clause to the SQL code with a fictitious table name using a random combination of 1-3 column names, and filter out any examples that could not be parsed by mo-sql-parsing\(^2\). Next, we train a SQL-to-Text model (T5-3B) on the Spider (Yu et al., 2019b) dataset in order to generate natural language statements for each SQL query.

### 3.2 Objectives for Multi-Task Pretraining

**MLM-Based Objectives** A critical component in pretraining encoder-decoder models is a MLM-based objective. In STAMP we follow the span corruption style of MLM from Raffel et al. (2020), which involves replacing contiguous whole words above some fixed threshold as latent human supervision.

---

\(^1\)As discussed in Appendix A, we consider accepted answers, favorite answers, or answers that received upvotes

\(^2\)https://github.com/klanhakoski/mo-sql-parsing
from the text with sentinel tokens in the inputs, and then the decoder generates the replaced text preceded by the corresponding to sentinel token. We set the mean span to 3, with a denoising rate of 15% following the default T5 configuration. This span corruption objective is applied to sequences of NL and SQL code. For pretraining datasets that also include structured knowledge we apply the masked column recovery (MCR) objective, as introduced in Yin et al. (2020), which encourages the model to learn table schemas using the natural language statement and row information as context. In our implementation, 25% of the column names and data types (when available) are masked with a sentinel token. Note, only MCR is applied to the sequence containing the column names to avoid overlapping MLM and MCR masking. More concretely, let $x_{\text{mask}} = (x_{\text{MLM}}, x_{\text{MCR}})$ be the input sequence combining MLM and MCR masking, then our masked span prediction loss $L_M$ over a sequence of length $T$ is:

$$L_M(\theta) = T \sum_{t=1}^{T} -\log P_\theta (y_t | z, y_{<t}),$$

where $z = x_{<S}$ is the left context and $y = x_{S\leq t}$ the right output.

**Combining Objectives** Prior work shows the importance of MLM (Liu et al., 2019; Aroca-Ouellette and Rudzicz, 2020; Raffel et al., 2020) and the benefit of including a small percentage of context-to-output objectives. For instance, Tay et al. (2022) recommend approximately 20% of the objective mixture to be context-to-output. However, unlike Tay et al. (2022) we are not pretraining from scratch, rather we seek to build on existing checkpoints and hence we consider greater rates of context-to-output. In our implementation, we sample an objective per-example during pretraining, where the pool of objectives depends on the data source of each example. Hence, each training mini-batch combines examples from multiple data sources that are formatted as a mix of objectives. Figure 2 summarizes our dataset and objective mix, showing the connection between each input data source and a corresponding objective.

**Context-to-Output Objectives** In addition to MLM-based objectives we include causal language modeling objectives (Radford et al., 2019; Liu et al., 2018), which partition sequences into contexts and outputs in order to mimic the format of many downstream tasks. For unimodal datasets, such as GitHub SQL, we create the context and output by uniformly sampling a split point based on line-breaks within each code example. For tabular datasets we treat the table as input and the paired NL as output, thereby teaching the model to connect the structured and unstructured information.

For Stack Overflow, the natural partition between a question and each of the answers defines the context to output splits. We use the augmentations described in 3.1 to create additional unique question-to-answer pairs. We apply dual learning to better align the question prompt with the answer.

Finally, for trimodal data like our augmented-TAPEX we model Table + NL $\rightarrow$ SQL, or in the dual learning (Wang et al., 2021b) setting we model Table + SQL $\rightarrow$ NL. Thus for a sequence $x$ of length $T$ with a split point $S \in (0, T)$ that is either randomly selected or based a natural split in the data, we define the context-to-output loss $L_{C2O}$ as:

$$L_{C2O}(\theta) = \sum_{t=S}^{T} -\log P_\theta (y_t | z, y_{<t}),$$

where $z = x_{<S}$ is the left context and $y = x_{S\leq t}$ the right output.

**Unified Format for Learning from Structured Knowledge**

In order to bridge the gap between pretraining and downstream tasks, we explore unified formats for structured knowledge. Connecting NL to structured knowledge is challenging with limited data. A unified table format, however, allows the model to leverage learning from large scale pretraining.
for smaller datasets. Moreover, in some cases Xie et al. (2022) report worse performance for multi-task versus single-task training, which we suspect is due to inconsistent formatting. Thus, we linearize structured knowledge into both row- and column-centric formats. Figure 3 shows the row-centric format, and Figure 4 shows the equivalent information in the column-centric format.

Lastly, we use special tokens in the encoder to preface each data modality (NL, structured knowledge, and SQL), and encourage sharing across tasks with common modalities. Additional tags prompt the decoder with the desired task, reflecting each of our objectives: MLM, table-to-text, SQL-to-SQL, Table and NL-to-SQL, Stack Overflow question answering, and dual learning variations.

4 Experiments

4.1 Evaluation Setup

We evaluate our pretrained checkpoints on SQL tasks following the UnifiedSKG framework (Xie et al., 2022). Specifically, for text-to-SQL benchmarking we evaluate on Spider without database row information (Yu et al., 2019b) and WikiSQL with row information (Zhong et al., 2017), as well as conversational text-to-SQL datasets SPaRC (Yu et al., 2019c) and CoSQL (Yu et al., 2019a), and in alignment with our bimodal objectives we also evaluate on SQL2Text (Shu et al., 2021). For each dataset we use pre-defined train, validation, and test splits. In Appendix C lists our evaluation settings, Appendix D contains details on the evaluation datasets, and Appendix E includes additional results.

4.2 Main Results

We present our main results in Table 1, with baseline results as reported in each comparison approach. We group models with SQL-specific decoders on top, and encoder-decoders like STAMP that have more general token decoders on bottom. Overall we find that our STAMP yields better results than domain specific solutions and text- or code-only pretrained models. SMBOP + GRAPPA (Rubin and Berant, 2021) is similar to our work with multi-task learning and additional pretraining, however they rely on a SQL specific parsing algorithm. Whereas, our framework focuses on larger, more diverse sources of structured knowledge and a complementary multi-task learning strategy.

We highlight that pretraining on structured information alone like TABERT (Yin et al., 2020), or a general code pretraining dataset like CodeT5 (Wang et al., 2021b) does not produce exceptional results on text-to-SQL. Likewise, a large multi-task learning approach like T0 performs worse than STAMP models and vanilla T5, indicating that the benefits of multi-task learning depend on having a degree of domain relevance. Specifically T0’s multi-task learning approach, which centers on text-only domains, does not benefit SQL tasks. Lastly, despite constrained decoding being very different than our approach, we include results for PICARD (Scholak et al., 2021) because it is an extremely effective approach that complements STAMP.

4.3 Ablation Studies

Denoising versus Context-to-Output In Table 2 we report development set performance of STAMP models that build on the T5-base checkpoint. We train each model on our full row-centrically orientated dataset and only vary the objective mixture. Unlike prior work (Tay et al., 2022; Aroca-Ouellette and Rudzicz, 2020) that pretrains from scratch, during our additional structured knowledge pretraining we observe that higher rates context-to-output objectives tend to perform best.

At the extremes of the objective mix we see mixed results. Setting MLM / context-to-output ratios to 100% / 0%, improves performance on text-to-SQL — indicating the benefit from our pretraining data. However, on the other extreme, model performance suffers with no MLM and only context-to-output. Nonetheless, by combining the two objectives we see the best performance overall. Specifically, an equal mix of MLM and completion either throughout pretraining or after one epoch of entirely MLM training results in noticeably higher performance compared to vanilla T5.

Our results complement those in literature (Tay et al., 2022; Wang et al., 2021b; Aghajanyan et al., 2021; Aribandi et al., 2021; Sanh et al., 2022; FitzGerald et al., 2022), showing the importance of mixing additional objectives with MLM. Unlike Tay et al. (2022), however, our results show that higher rates of context-to-output are optimal, which we attribute to our approach of building on prior checkpoints and not pretraining from scratch.

Tables versus SQL Datasets Table 3 presents an ablation study comparing of STAMP and CodeSTAMP models trained on different pretrain-
Table 1: Development set performance on text-to-SQL benchmarks for both T5, CodeT5, and our results with additional pretraining on our structured knowledge. All STAMP checkpoints train with a 50/50 mixture of context-to-output and MLM-based objectives. STAMP results are separated by variations in the pretraining data, specifically CC and RC denote column- and row-centric table formats, respectively, and w/ Tables denotes the full pretraining dataset whereas SQL-only is a subset that omits the NL+Table datasets. Note: A dagger (†) indicates constrained decoding approach, which is complementary but not used in our work, models in italics are our work.

Table 2: Development set performance for T5-base, and base-sized STAMP models pretrained on our full row-centric dataset with varying objective mixes. For each pretrained STAMP model we specify the proportion of training examples using the MLM-based objective, with the remaining examples using a dataset-specific context-to-output objective. We also explore dynamic mixing ratios, where 100→50% represents training with 100% MLM in the first epoch, followed by a 50%/50% mix of during the remaining epochs. 

Row-Centric versus Column-Centric. We pre-process the pretraining and benchmark datasets from UnifiedSKG (Xie et al., 2022) with consistent table formatting. Row-centric formats are more similar to natural language and do not require learning any new special tokens, which better leverages the original NL pretraining of T5. Whereas, the column-centric format requires special tokens that preface the table, columns, and each value in a column. While new special tokens must be learned from scratch, we hypothesized that the column-centric format is advantageous since text-to-SQL is inherently more column and schema oriented and often not dependent on row information. Surprisingly, Table 3 shows no clear advantage for either RC or CC formats. In fact, the mixed results hold for even across model sizes (Large vs Base) and initial pretraining (T5 vs CodeT5). Our results suggest that further pretraining on enough high-quality data helps to nullify the advantages or disadvantages of each table linearization method.

T5 versus CodeT5 as Starting Point. Table 3 shows the high performance of base-sized CodeT5 et al. (2020), our results show that adding SQL code to the data mix further boosts performance.
### Table 3: Development set performance on SQL benchmarks for both the original T5-base, T5-large, CodeT5-base, and CodeT5-large checkpoints, as well as our results with additional pretraining on our structured knowledge pretraining dataset. All STAMP checkpoints train with a 50/50 mixture of context-to-output and MLM-based objectives. STAMP results are separated by variations in the pretraining data, specifically CC and RC denote column- and row-centric table formats, respectively, and w/ Tables denotes the full pretraining dataset described in 3 whereas the SQL-only subset omits the Text+Table datasets. The best performer at each model size is shown in **bold**.

<table>
<thead>
<tr>
<th>Starting Checkpoint</th>
<th>Additional STAMP Pretraining Data</th>
<th>Spider (Exec ↑)</th>
<th>Sup. WikiSQL (EM ↑)</th>
<th>SParC (EM ↑)</th>
<th>CoSQL (EM ↑)</th>
<th>SQL2Text (BLEC ↑)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T5-Large</td>
<td>—</td>
<td>71.7</td>
<td>75.3</td>
<td>57.4</td>
<td>48.8</td>
<td>93.4</td>
</tr>
<tr>
<td>T5-Large</td>
<td>RC, w/ Tables</td>
<td>74.4</td>
<td>78.9</td>
<td><strong>61.4</strong></td>
<td><strong>53.7</strong></td>
<td>93.0</td>
</tr>
<tr>
<td>T5-Large</td>
<td>RC, SQL-only</td>
<td>72.8</td>
<td>79.5</td>
<td>60.1</td>
<td>51.4</td>
<td><strong>93.6</strong></td>
</tr>
<tr>
<td>T5-Large</td>
<td>CC, w/ Tables</td>
<td><strong>76.3</strong></td>
<td>79.3</td>
<td>59.6</td>
<td>51.4</td>
<td>93.3</td>
</tr>
<tr>
<td>T5-Large</td>
<td>CC, SQL-only</td>
<td>74.5</td>
<td>79.1</td>
<td>51.9</td>
<td>50.9</td>
<td>93.3</td>
</tr>
<tr>
<td>CodeT5-Large</td>
<td>—</td>
<td>68.4</td>
<td>76.6</td>
<td>57.9</td>
<td>48.4</td>
<td>91.9</td>
</tr>
<tr>
<td>CodeT5-Large</td>
<td>RC, w/ Tables</td>
<td>71.9</td>
<td>84.4</td>
<td>59.7</td>
<td>50.9</td>
<td>92.1</td>
</tr>
<tr>
<td>CodeT5-Large</td>
<td>CC, w/ Tables</td>
<td>72.8</td>
<td><strong>84.7</strong></td>
<td>58.7</td>
<td>52.0</td>
<td>92.1</td>
</tr>
<tr>
<td>T5-Base</td>
<td>—</td>
<td>60.8</td>
<td>74.1</td>
<td>49.9</td>
<td>42.4</td>
<td>93.7</td>
</tr>
<tr>
<td>T5-Base</td>
<td>RC, w/ Tables</td>
<td>64.5</td>
<td>77.9</td>
<td>51.9</td>
<td>44.5</td>
<td>93.2</td>
</tr>
<tr>
<td>T5-Base</td>
<td>RC, SQL-only</td>
<td>61.7</td>
<td>77.8</td>
<td>52.4</td>
<td>42.8</td>
<td>93.4</td>
</tr>
<tr>
<td>T5-Base</td>
<td>CC, w/ Tables</td>
<td>60.5</td>
<td>79.5</td>
<td>49.9</td>
<td>41.3</td>
<td>93.9</td>
</tr>
<tr>
<td>T5-Base</td>
<td>CC, SQL-only</td>
<td>59.2</td>
<td>79.5</td>
<td>46.8</td>
<td>38.9</td>
<td><strong>94.0</strong></td>
</tr>
<tr>
<td>CodeT5-Base</td>
<td>—</td>
<td>67.1</td>
<td>76.0</td>
<td>54.4</td>
<td>47.2</td>
<td>93.5</td>
</tr>
<tr>
<td>CodeT5-Base</td>
<td>RC, w/ Tables</td>
<td>69.0</td>
<td>83.5</td>
<td><strong>55.6</strong></td>
<td><strong>47.7</strong></td>
<td>92.9</td>
</tr>
<tr>
<td>CodeT5-Base</td>
<td>CC, w/ Tables</td>
<td><strong>69.2</strong></td>
<td><strong>84.5</strong></td>
<td>54.7</td>
<td>46.9</td>
<td>93.4</td>
</tr>
</tbody>
</table>

and CodeSTAMP models. Relative to their T5 and STAMP counterparts, the base-sized CodeT5 and CodeSTAMP models show significant performance gains across all text-to-SQL benchmarks. In particular, models based on the CodeT5-base checkpoint show exceptional performance when given row information in the tables, as is the case for WikiSQL. Interestingly, models based on CodeT5 do not exhibit the same performance gains compared to those based on T5 for large-sized models. In fact, models based on CodeT5-large only excel at WikiSQL, whereas models based on T5-large excel in all other tasks. We hypothesize that large-sized models based on CodeT5 do not outperform their peers in the same way as the base-sized models due to scaling issues caused by CodeT5’s much smaller CodeSearchNet (Husain et al., 2020) pretraining dataset, especially when using a smaller dataset to train the larger model. Additionally, we see that models based on CodeT5 checkpoints tend to perform worse on SQL2Text, which is likely because natural language in CodeT5’s original pretraining data is limited to comments in code, and hence the ability to generate natural language may be underdeveloped relative to T5.

### 5 Conclusion

We present STAMP, a pretraining framework for encoder-decoders on SQL tasks. We introduce a large scale pretraining dataset of tables, SQL code, discussions on Stack Overflow, and a modified TAPEX dataset (Liu et al., 2022). We complement our data with a multi-task learning framework to align the data modalities, finding that an equal mix of the objectives is optimal. We explore both row- and column-centric approaches to linearizing tables, creating a unified format across training stages. A column-centric format is often superior, challenging the conventional row-centric approach. Lastly, while PL pretraining may help generalization (Athiwaratkun et al., 2022), STAMP models based on T5 yield better performance.

---

3Our results for T5-Large on Spider, SParC, and CoSQL differ from Xie et al. (2022) and Scholak et al. (2021). On Spider we achieve 3.4%-points higher than Xie et al. (2022), and 4.5%-points higher than Scholak et al. (2021). In our implementation we use a maximum input sequence length of 1024 and an output sequence lengths of 256 to avoid truncation.
6 Limitations

While our work displays many strengths, we highlight some important limitations in our analysis. Namely, we pretrain our STAMP models on a range of sources containing structured knowledge, however our analysis is limited to text-to-SQL tasks and does not demonstrate if such pretraining helps more generally in structured information tasks. For instance, STAMP pretrains on tables with (1) masked column recovery as a way to learn the structure of a table using the rows and natural language statement as context, and (2) a context-to-output objective that always includes the table in the context (when available) — since this matches the format of text-to-SQL tasks. It is unclear if our objective choices for pretraining on tables perform equally well on the range of structured knowledge tasks, such as table question-answering, table summarization, data-to-text, fact verification, and others explored in Xie et al. (2022). Second, we acknowledge that significant GPU resources are required for pretraining, even in continued pretraining approaches like ours which limit the breadth of ablations studies. Conversely, our work explores pretraining at smaller scales where certain phenomena like strong zero-shot performance is unlikely. Pretraining specifically on structured knowledge has an unknown value at larger scales with models having tens or hundreds of billions of parameters.

7 Ethics Statement

We acknowledge the importance of the ACL Ethics Policy and agree with it. Large language models can appear confident while providing false information. In our work we are fortunate that incorrect SQL output is verifiable and take care to report the true reliability of the systems. Additionally we acknowledge that large language models, such as those studied in this work, may generate toxic language (Gehman et al., 2020). While we avoid pretraining on data sources and content from web domains with offensive language, we acknowledge that even our data gathered from reputable publishers introduces bias (Bolukbasi et al., 2016).

Acknowledgements

We would like to thank Henry Zhu for providing a sql-to-text model that we used to augment TAPEX with natural language statements.

References


Ewa Andrejczuk, Julian Martin Eisenschlos, Francesco Piccinno, Syrine Krichene, and Yasemin Altun. 2022. Table-to-text generation and pre-training with tabt5.


Jifan Chen, Yuhao Zhang, Lan Liu, Rui Dong, Xinchi Chen, Patrick Ng, William Yang Wang, and Zhiheng Huang. 2022. Improving Cross-task Generalization
of Unified Table-to-text Models with Compositional Task Configurations.

Wenhu Chen, Hongmin Wang, Jianshu Chen, Yunkai Zhang, Hong Wang, Shiyang Li, Xiyou Zhou, and William Yang Wang. 2020. TabFact: A Large-scale Dataset for Table-based Fact Verification.


Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: Pretraining Text Encoders as Discriminators Rather Than Generators.


Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. 2017. Don’t Decay the Learning Rate, Increase the Batch Size.


A.1 Stack Overflow Augmentations

We perform several augmentation steps on Stack Overflow examples to construct our pretraining dataset. Our first step is to create four augmented versions of each question using random word deletion, random word appending, synonym replacement, and paraphrasing. Next, we create up to five different combinations of input-label pairs by re-arranging the answers.

For some pertinent background on Stack Overflow, each example consists of a question and one or more answers. The user who answered the question can mark the answer that solved their problem as correct, and other users can upvote answers that they found useful as well.

Let \( N \) be the number of answers for a question. The following strategies are used to create the labels for the augmented examples:

1. The accepted answer (if there is one)
2. The most upvoted answer if it has been upvoted more than the accepted answer.
3. Concatenation of all answers

4. Randomly select an answer $A_i$ and append all answers up to and including that one to the question, then use the concatenation of all $A_{i+1}, A_{i+2} \ldots A_N$ answers as the label

5. Randomly select an answer, $A_i$, and append all answers up to and including that one to the question. Randomly select another answer, $A_k$, from the remaining $A_{i+1}, A_{i+2} \ldots A_N$ answers and use the concatenation of all $A_k, A_{k+1} \ldots A_N$ answers as the label

Each of these strategies is constrained by a total sequence length of 1024 tokens. If we need to truncate any tokens, we truncate them in the following order:

1. Text in Answer
2. Code in Question
3. Text in Question

Our intuition is that this is the order of least important to most important to preserve the logical relationship between question and answer, with code in the answer being the most critical (which is never truncated).

A.2 Data Filtering

As briefly mentioned in 3.1, we filter noisy examples from both the table and SQL dataset. Below we provide more details on this pre-processing step.

Tabular Filtering Since table data is often web-scraped it contains many noisy examples. Specifically, examples where the table information has a tenuous relation to the paired natural language statement. Moreover, since our initial collection of raw data was much larger for table sources versus SQL source, we chose to implement a filtering approach to reduce these noisy examples. Specifically, we first calculate the edit-similarity between each sample’s table and the NL statement, after removing special tokens or tags. We then compute the same metric on ToTTo, which is a high-quality table-to-text benchmark, and qualitatively chose our filtering threshold as 50.0 which is slightly lower than ToTTo’s average edit-similarity. All samples from our Wiki, Web, and ArXiv tables datasets with an edit-similarity below 50.0 are removed. In total we remove approximately 74% of samples from the raw data.

Github SQL Filtering For the Github SQL data we again see a large proportion of noisy or repetitive samples in the raw data. Specifically, Github SQL data can contain many repetitive statements within one sample, such as thousands of consecutive INSERT statements that data into a table. The insert statements are often either very repetitive, or contain very noisy information like compressed images, PDFs, or spatial objects. Our filtering method largely consists of using regular expression to identify such repetitive statements. After finding long sequences of insert statements we keep only a random sample of 10 insert statements if the insert statements are repetitive but not overly long or unreadable. However, we remove all insert statements that load noisy information into a table. In total the number of samples staying approximately the same, however we reduce the size of the dataset by approximately 61%.

A.3 Pretraining Dataset Statistics

In Table 4 we provide summary statistics for the pretraining dataset, including each of the SQL and Table subsets. Raw document counts help to show the amount of filtering applied to the raw data in order reduce noisy and potentially detrimental samples, whereas the final training sample counts show the training dataset size after tokenizing and partitioning documents into sequences.

B Pretraining Hyperparameters

Batch size. For 3B and large models we train for at a small batch size of 64 for the first epoch, then for most of the second and third epoch we double the batch size to 128, and then for the final 5-10% of training we double the batch size again to 256. Starting with a small batch size provides better gradient efficiency, while larger batch sizes give more precise gradient estimates which is beneficial later in training (Smith et al., 2017). For base sized models we opt for a batch size of 128 for all three epochs before the cooldown period.

Sequence length. Data are pre-processed and tokenized offline into sequences of at most 1024 tokens. We do not pack inputs, and instead use one example in per input and then pad accordingly. For the larger T5-3B model we found that training for the first 75-90% of steps on data pre-processed into a shorter max sequence length of 768 or 896, and then the remainder of training on data with 1024 tokens provided improved computational efficiency.
without a discernible degradation in performance. Encoder inputs begin with a special token indicating the data modality, and the decoder inputs begin with a special token indicating the desired task. All sequences end with the same end of sequence token as Raffel et al. (2020).

Optimization. All models are pretrained with the AdamW (Kingma and Ba, 2015) optimizer, using an initial learning rate of $1e^{-4}$, and set momentum of $\beta_1 = 0.9$ and $\beta_2 = 0.98$. Our learning rate warms-up linearly over the first 1% of training steps, and then decays following a fixed cosine annealing schedule to $1e^{-7}$ after approximately 3 epochs. We set a gradient norm clipping with a maximum gradient norm of 1.0 (Pascanu et al., 2013). We train models based on T5 (Raffel et al., 2020) using the bf16 data type, whereas for models based on CodeT5 (Wang et al., 2021b) we use the fp16 data type in order to match the data type from STAMP-CC. For finetuning we follow the experimental setup of UnifiedSKG (Xie et al., 2022). Specifically, we use the Adafactor optimizer with decaying learning rate that is initially set to 5e-5, we set the batch size to 32, train for up to 200 epochs, and generate sequences using a beam size of 1. However, for WikiSQL we set a batch of 128, train for a maximum of 100 epochs, and use a beam size of 4. We use the same maximum lengths for the input and output as UnifiedSKG, except for Spider, SParC, and CoSQL where we increase input maximum length to 1024 and output to 256 sentence piece tokens to avoid truncating the inputs or outputs.

**C Evaluation Settings**

For finetuning we follow the experimental setup of UnifiedSKG (Xie et al., 2022). Specifically, we use the Adafactor optimizer with decaying learning rate that is initially set to 5e-5, we set the batch size to 32, train for up to 200 epochs, and generate sequences using a beam size of 1. However, for WikiSQL we set a batch of 128, train for a maximum of 100 epochs, and use a beam size of 4. We use the same maximum lengths for the input and output as UnifiedSKG, except for Spider, SParC, and CoSQL where we increase input maximum length to 1024 and output to 256 sentence piece tokens to avoid truncating the inputs or outputs.

<table>
<thead>
<tr>
<th>Data Source</th>
<th>Modalities</th>
<th>Num. Raw Documents</th>
<th>Num. Training Samples</th>
<th>Avg. Number of Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Initial (K)</td>
<td>Filtered (K)</td>
<td></td>
</tr>
<tr>
<td>Github SQL</td>
<td>SQL</td>
<td>1,026</td>
<td>1,019</td>
<td>1,918</td>
</tr>
<tr>
<td>Stack Overflow</td>
<td>NL, SQL</td>
<td>1,670</td>
<td>1,631</td>
<td>4,480</td>
</tr>
<tr>
<td>Aug. TAPEX</td>
<td>NL, Table, SQL</td>
<td>2,165</td>
<td>2,165</td>
<td>2,005</td>
</tr>
<tr>
<td>Wiki Tables</td>
<td>NL, Table</td>
<td>6,350</td>
<td>3,080</td>
<td>3,080</td>
</tr>
<tr>
<td>Web Tables</td>
<td>NL, Table</td>
<td>32,295</td>
<td>7,032</td>
<td>7,032</td>
</tr>
<tr>
<td>ArXiv Tables</td>
<td>NL, Table</td>
<td>119</td>
<td>22</td>
<td>24</td>
</tr>
<tr>
<td>Full Dataset</td>
<td>NL, Table, SQL</td>
<td>43,766</td>
<td>14,991</td>
<td>18,612</td>
</tr>
</tbody>
</table>

Table 4: STAMP Pretraining dataset statistics by source. After the raw documents are filtered, we create training examples by partitioning documents into sequences of 1024 tokens which can result in more training samples than the initial set of raw documents. In the case of Stack Overflow we also augment the data creating a much larger collection of training samples from the initial pool of documents. Note: Raw document counts and final number of training samples are listed in thousands (K), the final pretraining dataset contains 18,612,078 samples.

<table>
<thead>
<tr>
<th>Pretrained Model</th>
<th>Finetune Method</th>
<th>Spider (Exec ↑)</th>
<th>Sup. WikiSQL (EM ↑)</th>
<th>SParC (EM ↑)</th>
<th>CoSQL (EM ↑)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAMP-RC</td>
<td>STF</td>
<td>74.4</td>
<td>78.9</td>
<td>61.4</td>
<td>53.7</td>
</tr>
<tr>
<td>STAMP-RC</td>
<td>MTF</td>
<td>74.0</td>
<td>78.6</td>
<td>61.9</td>
<td>55.0</td>
</tr>
<tr>
<td>STAMP-CC</td>
<td>STF</td>
<td>76.3</td>
<td>79.3</td>
<td>59.6</td>
<td>51.4</td>
</tr>
<tr>
<td>STAMP-CC</td>
<td>MTF</td>
<td>73.9</td>
<td>79.1</td>
<td>61.3</td>
<td>54.2</td>
</tr>
<tr>
<td>CodeSTAMP-RC</td>
<td>STF</td>
<td>74.5</td>
<td>84.3</td>
<td>58.8</td>
<td>50.6</td>
</tr>
<tr>
<td>CodeSTAMP-RC</td>
<td>MTF</td>
<td>73.3</td>
<td>83.9</td>
<td>59.4</td>
<td>51.9</td>
</tr>
<tr>
<td>CodeSTAMP-CC</td>
<td>STF</td>
<td>72.8</td>
<td>84.7</td>
<td>58.7</td>
<td>52.0</td>
</tr>
<tr>
<td>CodeSTAMP-CC</td>
<td>MTF</td>
<td>71.3</td>
<td>83.5</td>
<td>58.3</td>
<td>50.8</td>
</tr>
</tbody>
</table>

Table 5: Development set performance on text-to-SQL benchmarks for large sized T5, STAMP CodeT5, and CodeSTAMP that are either Single-Task Finetuned (STF) or Multi-Task Finetuned (MTF) on all text-to-SQL datasets simultaneously. All STAMP checkpoints are pretrained with a 50/50 mixture of context-to-output and MLM-based objectives on the full pretraining dataset. STAMP results differentiated by whether they’re trained with column-CC or row-centric RC table formats. We highlight results where multi-task finetuning outperforms single-task finetuning on an equivalent model in bold.

**D Evaluation Datasets**

We evaluate our model on each of the aforementioned datasets using the standard metrics for each task. We use the standard train, validation, and test splits for each of the datasets.

**Spider** The Spider dataset has 10,181 question-query pairs with queries using 200 databases representing 138 different domains and tables that are joined via foreign keys. We use the standard training and development splits, where training, development, and test sets have a 7:1:2 ratio, and each database appears in only one set (Yu et al., 2019b).

**Fully Supervised WikiSQL** The WikiSQL dataset has 80,564 question-query pairs, involving over 30,000 tables from Wikipedia (Zhong et al.,...
Table 6: Average performance on SQL benchmarks over three finetuning runs with standard deviations. All STAMP checkpoints train with a 50/50 mixture of context-to-output and MLM-based objectives. STAMP results are separated by variations in the pretraining data, specifically CC and RC denote column- and row-centric table formats, respectively, and w/Tables denotes the full pretraining dataset whereas SQL-only is a subset that omits the NL+Table datasets. Note: A dagger (†) indicates datasets where only a development set is available for assessing variance in performance, and models in italics are our work.

E Additional Results

Single- versus Multi-Task Learning We explore the benefits of finetuning and evaluating either individually on each dataset (Single-Task Finetuning, STF) versus finetuning on all of the text-to-SQL benchmarks simultaneously then evaluating (Multi-Task Finetuning, MTF). For multi-task finetuning we balance the size of different datasets during training using the temperature up-sampling method proposed in Xie et al. (2022) and set the temperature to 2. The results of the ablation are presented in Table 5. We find mixed the results of multi-task finetuning. In almost every model MTF results in noticeably better performance on the conversational SQL datasets SParC and CoSQL, however results for Spider and WikiSQL are slightly worse. We suspect that the close similarity between SParC and CoSQL explains the mutual benefit of multi-task finetuning. On the other hand, Spider uses a schema-only input format, whereas WikiSQL includes database content and is typically less difficult than Spider.

Performance Confidence Intervals In Table 6 we report more a more detailed look at our main results. Specifically, we report the average performance of our models over three finetuning runs and list the standard deviation in the performances.