Abstract

Text-to-SQL parsers typically struggle with databases unseen during the train time. Adapting parsers to new databases is a challenging problem due to the lack of natural language queries in the new schemas. We present REFiLL, a framework for synthesizing high-quality and textually diverse parallel datasets for adapting a Text-to-SQL parser to a target schema. REFiLL learns to retrieve-and-edit text queries from the existing schemas and transfers them to the target schema. We show that retrieving diverse existing text, masking their schema-specific tokens, and refilling with tokens relevant to the target schema, leads to significantly more diverse text queries than achievable by standard SQL-to-Text generation methods. Through experiments spanning multiple databases, we demonstrate that fine-tuning parsers on datasets synthesized using REFiLL consistently outperforms the prior data-augmentation methods.

1 Introduction

Natural Language Interface to Databases (NLIDB) that translate text queries to executable SQLs is a challenging task in the field of Semantic Parsing (Zelle and Mooney, 1996; Zettlemoyer and Collins, 2005; Berant et al., 2013). In addition to understanding the natural language and generating an executable output, Text-to-SQL also requires the ability to reason over the schema structure of relational databases. Recently, datasets such as Spider (Yu et al., 2018) comprising of parallel (Text,SQL) pairs over hundreds of schemas have been released, and these have been used to train state-of-the-art neural Text-to-SQL models (Wang et al., 2020; Scholak et al., 2021a; Rubin and Berant, 2021; Scholak et al., 2021b; Xu et al., 2021). However, several studies have independently shown that such Text-to-SQL models fail catastrophically when evaluated on unseen schemas from the real-world databases (Suhr et al., 2020; Lee et al., 2021; Hazoom et al., 2021). Adapting existing parsers to new schemas is challenging due to the lack of parallel data for fine-tuning the parser.

Synthesizing parallel data, that is representative of natural human generated queries (Wang et al., 2015; Herzig and Berant, 2019), is a long-standing problem in semantic parsing. Several methods have been proposed for supplementing with synthetic data, ranging from grammar-based canonical queries to full-fledged conditional text generation models (Wang et al., 2015; Herzig and Berant, 2019; Zhong et al., 2020; Yang et al., 2021; Zhang et al., 2021; Wang et al., 2021). For Text-to-SQL, data-augmentation methods are primarily based on training an SQL-to-Text model using labeled data from pre-existing schemas, and generating data in the new schemas. We show that the text generated by these methods, while more natural than canonical queries, lacks the rich diversity of natural multi-user queries. Fine-tuning with such data often deteriorates the model performance since the lack of diversity leads to a biased model.

We propose a framework called REFiLL (§ 2) for generating diverse text queries for a given SQL workload that is often readily available (Baik et al., 2019). REFiLL leverages parallel datasets from several existing schemas, such as Spider (Yu et al., 2018), to first retrieve a diverse set of text paired with SQLs that are structurally similar to a given SQL $q$ (§ 2.1). Then, it trains a novel schema translator model for converting the text of the training schema to the target schema of $q$. The schema translator is decomposed into a mask and fill step to facilitate training without direct parallel examples of schema translation. Our design of the mask module and our method of creating labeled data for the fill module entails non-trivial details that we explain in this paper (§ 2.2). REFiLL also incorporates a method of filtering-out inconsistent (Text,SQL) pairs using an independent binary classifier (§ 2.3), that provides more useful
quality scores, than the cycle-consistency based filtering (Zhong et al., 2020). Our approach is related to retrieve-and-edit models that have been used for semantic parsing (Hashimoto et al., 2018), dialogue generation (Chi et al., 2021), translation (Cai et al., 2021), and question answering (Karpukhin et al., 2020). However, our method of casting the “edit” as a two-stage mask-and-fill schema translation model is different from the prior work.

We summarize our contributions as follows: (i) We propose the idea of retrieving and editing natural text from several existing schemas for transferring it to a target schema, obtaining higher text diversity compared to the standard SQL-to-Text generators. (ii) We design strategies for masking schema-specific words in the retrieved text and training the ReFILL model to fill in the masked positions with words relevant to the target schema. (iii) We filter high-quality parallel data using a binary classifier and show that it is more efficient than existing methods based on cycle-consistency filtering. (iv) We compare ReFILL with prior data-augmentation methods across multiple schemas and consistently observe that fine-tuning Text-to-SQL parsers on data generated by ReFILL leads to more accurate adaptation.

2 Diverse data synthesis with ReFILL

Our goal is to generate synthetic parallel data to adapt an existing Text-to-SQL model to a target schema unseen during training. A Text-to-SQL model \( \mathcal{M} : \mathcal{X} \times \mathcal{S} \rightarrow \mathcal{Q} \) maps a natural language question \( x \in \mathcal{X} \) for a database schema \( s \in \mathcal{S} \), to an SQL query \( q \in \mathcal{Q} \). We assume a Text-to-SQL model \( \mathcal{M} \) trained on a dataset \( \mathcal{D}_{\text{train}} = \{ (x_i, s_i, q_i) \}_{i=1}^{N} \) consisting of text queries \( x_i \) for a database schema \( s_i \), and the corresponding gold SQLs \( q_i \). The train set \( \mathcal{D}_{\text{train}} \) typically consists of examples from a wide range of schemas \( s_i \in \mathcal{S}_{\text{train}} \). For example, the Spider dataset (Yu et al., 2018) contains roughly 140 schemas in the train set. We focus on adapting the model \( \mathcal{M} \) to perform well on a target schema \( s \) different from the training schemas in \( \mathcal{S}_{\text{train}} \). To achieve this, we present a method of generating synthetic data \( \mathcal{D}_{\text{syn}} \) of Text-SQL pairs containing diverse text queries for the target schema \( s \). We fine-tune the model \( \mathcal{M} \) on \( \mathcal{D}_{\text{syn}} \) to adapt it to the schema \( s \). Our method is agnostic to the exact model used for Text-to-SQL parsing. We assume that on the new schema \( s \) we have a workload \( \mathcal{QW}_s \) of SQL queries. Often in existing databases a substantial SQL workload is already available in the query logs at the point a DB manager decides to incorporate the NL querying capabilities (Baik et al., 2019). The workload is assumed to be representative but not exhaustive. In the absence of a real workload, a grammar-based SQL generator may be used (Zhong et al., 2020; Wang et al., 2021).

Figure 1 and Algorithm 1 summarizes our method for converting a workload \( \mathcal{QW}_s \) of SQL queries into a synthetic dataset \( \mathcal{D}_{\text{syn}} \) of Text-SQL pairs containing diverse text queries. Given an SQL query \( q \in \mathcal{QW}_s \) for the target schema \( s \), our method first retrieves related SQL-Text pairs \( \{ q_r, x_r \} \) \( r=1 \) from \( \mathcal{D}_{\text{train}} \) on the basis of a tree-edit-distance measure such that the SQLs \( \{ q_r \} \) \( r=1 \) in the retrieved pairs are structurally similar to the SQL \( q \) (§ 2.1). We then translate each retrieved text query \( x_r \) so that its target SQL changes from \( q_r \) to \( q \) on schema \( s \) (§ 2.2). We decompose this task into two steps: masking out schema-specific tokens in \( x_r \), and filling the masked text to make it consistent with \( q \) using a conditional text generation model \( \mathcal{B} \) like BART (Lewis et al., 2020). The translated text may be noisy since we do not have direct supervision to train such models. Thus, to improve the overall quality of the synthesized data we filter out the inconsistent SQL-Text pairs using an independent binary classifier (§ 2.3). Finally, we adapt the Text-to-SQL model \( \mathcal{M} \) for the target schema \( s \) by fine-tuning it on the diverse, high-quality filtered data \( \mathcal{D}_{\text{syn}} \) synthesized by ReFILL.

2.1 Retrieving related queries

Given an SQL \( q \in \mathcal{QW}_s \) sampled from SQL workload, we extract SQL-Text pairs \( \{ q_r, x_r \} \in \mathcal{D}_{\text{train}} \), from the train set such that the retrieved SQLs \( \{ q_r \} \) are structurally similar to the SQL \( q \). We utilize tree-edit-distance (Pawlik and Augsten, 2015, 2016) between the relational algebra trees of SQLs \( q \) and \( q_r \) — smaller distance implies higher structural similarity. Since the retrieved SQLs come

---

**Algorithm 1: Data Synthesis with ReFILL**

```
1 input: \( \mathcal{QW}_s, \mathcal{M}, \mathcal{D}_{\text{train}} \)
2 \( \mathcal{D}_{\text{syn}} \leftarrow \phi \\
3 for q \leftarrow \text{SampleSQLQueries}(\mathcal{QW}_s) \) do
4   \{ q_r, x_r \} \leftarrow \text{RetrieveRelatedPairs}(q, \mathcal{D}_{\text{train}})
5   \{ x'_r \} \leftarrow \text{MaskSchemaTokens}(\{ q_r, x_r \})
6   \{ x_r \} \leftarrow \text{EditAndFill}(\{ q, x'_r \})
7   \mathcal{D}_{\text{syn}} \leftarrow \mathcal{D}_{\text{syn}} \cup \text{Filter}(\{ q, x_r \})
8 \mathcal{M}_{\text{new}} \leftarrow \text{fine-tune}(\mathcal{M}, \mathcal{D}_{\text{syn}})
```
from different schemas, we modify the tree-edit-distance algorithm to ignore the schema names and the database values. The tree-edit-distance is further normalized by the size of the larger tree. We only consider the \( \{q_r, x_r\} \) pairs where the SQLs \( \{q_r\} \) have a distance of less than 0.1 w.r.t. the SQL \( q \). Within datasets like Spider that span hundreds of schemas, it is often possible to find several SQLs structurally similar to a given SQL \( q \). For example, in Spider we found that 76% of the train SQLs contain at least three zero-distance (structurally identical) neighbours in other schemas. In Figure 2, we present more detailed statistics.

### 2.2 Translating text of related queries

Our next goal is to translate the retrieved \( x_r \) from being a text for SQL \( q_r \) to a text \( \hat{x} \) for SQL \( q \), where \( q \approx q_r \) structurally. However, we do not have a readily labeled dataset to learn a model that translates \( x_r \) to \( \hat{x} \) while being consistent with \( q \). We therefore decompose this task into two steps: 1) A simpler task of masking schema-specific tokens in \( x_r \) to get a template \( x_r^{\text{masked}} \) and 2) A conditional text generation model that maps \( \{x_r^{\text{masked}}, q\} \) to the text \( \hat{x} \) consistent with \( q \), by filling the masked positions in \( x_r^{\text{masked}} \) as per \( q \). We re-purpose \( D_{\text{train}} \) to get indirect supervision for training the text generation model. We now present each step in detail.

#### Masking the retrieved text

Converting the retrieved text queries \( \{x_r\} \) to masked templates \( \{x_r^{\text{masked}}\} \) is a critical component of ReFill’s pipeline since irrelevant tokens like references to schema elements of the original database can potentially misguide the text generation module. Our initial approach was to mask tokens based on a match of text tokens with schema names and manually refined schema-to-text linked annotations as in Lei et al. (2020). However, this approach failed to mask all schema-related terms since their occurrences in natural text often differed significantly from schema names in the database. Table A7 shows some anecdotes. Consequently, we designed

![Figure 1: Diverse parallel data synthesis by retrieving-and-editing existing examples using ReFill.](image)

![Figure 2: Frequency distribution of average tree-edit-distance between SQLs and their three nearest neighbours from other schemas within Spider’s train set.](image)
a simple frequency-based method of masking that is significantly more effective for our goal of using the masked text to just guide the diversity. For each word that appears in the text queries of the train set, we count the number of distinct databases where that word gets mentioned at least once. For example, common words like (‘show’, ‘what’, ‘list’, ‘order’) get mentioned in more than 90% of the schemas, and domain specific words like (‘countries’, ‘government’) occur only in text queries of a few schemas. We mask out all the words that appear in less than 50% of the schemas. The words to be masked are replaced by a special token MASK, and consecutive occurrences of MASK are collapsed into a single MASK token. Thus we obtain masked templates \{x_i^{\text{masked}}\} retaining minimal information about their original schema.

**Editing and Filling the masked text** Given a masked template \(x_i^{\text{masked}}\), and an SQL query \(q\), we wish to edit and fill the masked portions in \(x_i^{\text{masked}}\) to make it consistent with the SQL \(q\). We utilize a conditional text generation model \(B\) like BART (Lewis et al., 2020) for this purpose. We first convert \(q\) into a pseudo-English representation \(q_i^{\text{Eng}}\) similar to Shu et al. (2021), to make it easier for \(B\) to encode \(q\). In addition, we wrap the table, column, or value tokens in \(q_i^{\text{Eng}}\) with special tokens to provide explicit signals to the text generation model \(B\) that such tokens are likely to appear in the output text \(\hat{x}\). Next, we concatenate the tokens in \(x_i^{\text{masked}}\) and \(q_i^{\text{Eng}}\) for jointly encoding them as an input to \(B\). The output of \(B\)’s decoder is text \(\hat{x}\), which is expected to be consistent with the SQL \(q\).

Since we do not have direct supervision to fine-tune \(B\) for this task, we present a method of re-purposing \(D_{\text{train}}\) for fine-tuning \(B\). \(D_{\text{train}}\) contains SQL-Text pairs \((q_i, x_i)\) from various schemas \(s_i\). A Naïve way to train \(B\) is to provide \([x_i^{\text{masked}}, q_i^{\text{Eng}}]\), the concatenation of \(x_i^{\text{masked}}\) and \(q_i^{\text{Eng}}\) as an input to the encoder and maximize the likelihood of \(x_i\) in the decoder’s output. This way the decoder of \(B\) learns to fill the masked tokens in \(x_i^{\text{masked}}\) by attending to \(q_i^{\text{Eng}}\) to recover \(x_i\) in the output. While useful for learning to refill the masked positions, this Naïve method of training \(B\) is mismatched from its use during inference in two ways: (i) For a given SQL \(q\), REFill might fail to retrieve a similar structure neighbour of \(q_i\) from \(D_{\text{train}}\). In such cases, \(B\) should be capable of falling back to pure SQL-to-Text generation mode to directly translate \(q\) into \(\hat{x}\). (ii) During inference, \(x_i^{\text{masked}}\) and \(q_i^{\text{Eng}}\) come from different schemas. However, during Naïve training, the masked text \(x_i^{\text{masked}}\) and the SQL \(q_i\) are derived from the same example \((q_i, x_i)\). To address these two limitations, we train \(B\) in a more Robust manner as follows: (a) For a random one-third of the train steps we train \(B\) in the Naïve way, allowing \(B\) to learn the filling of the masked tokens using \(q_i^{\text{Eng}}\). (b) For another one-third, we pass only \(q_i^{\text{Eng}}\) as an input and maximize the likelihood of \(x_i\). This ensures that model is capable of generating the text from the \(q_i^{\text{Eng}}\) alone, if the templates \(x_i^{\text{masked}}\) are unavailable or noisy. (c) For the remaining one-third, we first retrieve an SQL-Text pair \((q_j, x_j)\), from a different schema such that the SQL \(q_j\) is structurally similar to \(q_i\) (§ 2.1), and the word edit distance between the masked templates \(x_i^{\text{masked}}\) and \(x_j^{\text{masked}}\) is also small. We can then replace \(x_i^{\text{masked}}\) with \(x_j^{\text{masked}}\) and encode \([x_j^{\text{masked}}, q_i^{\text{Eng}}]\) as an input to \(B\) and maximize the likelihood of \(x_i\) in the decoder’s output. This step makes the training more consistent with the inference, as \(x_i^{\text{masked}}\) and \(q_i^{\text{Eng}}\) now come from different schemas. In § 5.4, we justify training Robustly compared to Naïve training.

### 2.3 Filtering the Generated Text
Since the data synthesized using REFill is used to fine-tune a downstream Text-to-SQL parser, we learn a Filtering model \(F: (X, Q) \rightarrow \mathbb{R}\) to discard inconsistent examples from the generated dataset. \(F\) assigns lower scores to inconsistent Text-SQL pairs. For each SQL \(q \in QW_s\), we select the top-5 sentences generated by REFill and discard all the sentences that are scored below a fixed threshold as per the filtering model. Existing work depended on a trained Text-to-SQL parser \(M\) to assign cycle-consistency scores (Zhong et al., 2020). However, we show that cycle-consistency filtering favors text on which \(M\) already performs well, and hence does not result in a useful dataset for fine-tuning \(M\).

We instead train a filtering model \(F\) as a binary classifier, independent of \(M\). The Text-SQL pairs \(\{(x_i, q_i)\}\) in the training set \(D_{\text{train}}\), serve as positive (consistent) examples and we synthetically generate the negative (inconsistent) examples as follows: (i) Replace DB values in the SQL \(q_i\) with arbitrary values sampled from the same column of the database. (ii) Replace SQL-specific tokens in \(q_i\) with their corresponding alternates e.g. replace ASC with DESC, or ‘>’ with ‘<’. (iii) Cascade previous two perturbations. (iv) Replace the entire SQL \(q_i\) with a randomly chosen SQL \(q_j\) from the same
schema. (v) Randomly drop tokens in the text query \( x_i \) with a fixed probability of 0.3. (vi) Shuffle a span of tokens in the text query \( x_i \), with span length set to 30% of the length of \( x_i \). Thus, for a given Text-SQL pair \((x_i, q_i)\) we obtain six corresponding negative pairs \( \{(x_{i}^{n}, q_{i}^{n})\}_{j=1}^{6} \). Let \( s_i \) be the score provided by the filtering model for the original pair \((x_i, q_i)\) and \( \{s_j\}_{j=1}^{6} \) be the scores assigned to the corresponding negative pairs \( \{(x_{j}^{n}, q_{j}^{n})\}_{j=1}^{6} \). We supervise the scores from the filtering model using a binary-cross-entropy loss over the Sigmoid activations of scores as in Equation 1.

\[
\mathcal{L}_{\text{bce}} = -\log \sigma(s_i) - \sum_{j=1}^{6} \log \sigma(1 - s_j) \tag{1}
\]

To explicitly contrast an original pair with its corresponding negative pairs we further add another Softmax-Cross-Entropy loss term.

\[
\mathcal{L}_{\text{xent}} = -\log \frac{\exp(s_i)}{\exp(s_i) + \sum_{j=1}^{6} \exp(s_j)} \tag{2}
\]

3 Related Work

**SQL-to-Text generation** Many prior works perform training data augmentation via pre-trained text generation models that translate SQLs into natural text (Guo et al., 2018; Zhong et al., 2020; Shi et al., 2020; Zhang et al., 2021; Wang et al., 2021; Yang et al., 2021; Shu et al., 2021). For example, Wang et al. (2021) fine-tune BART (Lewis et al., 2020) on parallel SQL-Text pairs to learn an SQL-to-Text translation model. Shu et al. (2021) propose a similar model that is trained in an iterative-adversarial way along with an evaluator model. The evaluator learns to identify inconsistent SQL-Text pairs, similar to our filtering model. To retain high quality synthesized data Zhong et al. (2020) additionally filter out the synthesized pairs using a pre-trained Text-to-SQL model based on cycle consistency, that we show to be sub-optimal (§ 5.5).

The SQL workload in most of the prior work was typically sampled from hand-crafted templates or a grammar like PCFG induced from existing SQLs, or crawling SQLs from open-source repositories Shi et al. (2020). However, database practitioners have recently drawn attention to the fact that SQL workloads are often pre-existing and should be utilized (Baik et al., 2019).

**Retrieve and Edit Methods** Our method is related to the retrieve-and-edit framework, which has been previously applied in various NLP tasks. In Semantic Parsing, question and logical-form pairs from the training data relevant to the test-input question are retrieved and edited to generate the output logical forms in different ways (Shaw et al., 2018; Das et al., 2021; Pasupat et al., 2021; Gupta et al., 2021). In machine translation, memory augmentation methods retrieve-and-edit examples from translation memory to guide the decoder’s output (Hossain et al., 2020; Cai et al., 2021). Our editing step — masking followed by refilling is similar to style transfer methods that minimally modify the input sentence with help of retrieved examples corresponding to a target attribute (Li et al., 2018). In contrast to learning a retriever, we find simple tree-edit distance to be an effective metric for retrieving the relevant examples for our task.

4 Experimental Set-up

We adapt pretrained Text-to-SQL parsers on multiple database schemas unseen during the train time. Here, we describe the datasets, models, and evaluation metrics used in our experiments.

**Datasets**: We primarily experiment with the Spider dataset (Yu et al., 2018). Spider’s train split contains 7000 Text-to-SQL examples spanning 140 database schemas, and the dev split contains roughly 1000 examples spanning 20 schemas. Since individual schemas in the dev split typically contain less than 50 examples, to evaluate on a larger set of examples we adapt and evaluate the Text-to-SQL parser on groups of similar schemas instead of individual schemas. We create 4 groups, with each group having database schemas from a similar topic. For example, Group-1 consists of databases {Singer, Orchestra, Concerts}. We utilize all the available Text-SQL pairs in each group for evaluation. In appendix Table A.1, we provide detailed statistics about each group. On average, each group contains 69 unique SQLs and 131 evaluation examples. To simulate a query workload \( QW_s \) for each group, we randomly select 70% of the available SQLs and replace the constant-values in the SQLs with values sampled from their corresponding column in the database. We also evaluate on query workloads of size 30% and 50% of the available SQL queries. The SQL queries in the workload are translated using Refill or an SQL-to-Text model, and the resulting Text-SQL pairs are

---

1 Code: github.com/awasthiabhijeet/refill

2Spider’s test-split is inaccessible as of 10/26/2022
then used to fine-tune a base Text-to-SQL parser.

We further experiment with four datasets outside Spider in Section 5.6. We work with GeoQuery (Zelle and Mooney, 1996), Academic (Li and Jagadish, 2014), IMDB and Yelp (Navid Yaghmazadeh and Dillig, 2017). We utilize the pre-processed version of these datasets open-sourced by Yu et al. (2018). In appendix Table A2, we present statistics about each of the four datasets.

Text-to-SQL parser: We experiment with SMBOP (Rubin and Berant, 2021) as our base Text-to-SQL parser, and utilize author’s implementation. The SMBOP model is initialized with a ROBERTA-BASE model, followed by four RAT layers, and trained on the train split of Spider dataset. The dev set used used for selecting the best model excludes data from the four held-out evaluation groups.

Edit and Fill model: We utilize a pre-trained BART-BASE as our conditional text generation model for editing and filling the masked text. The model is fine-tuned using the train split of Spider dataset as described in Section 2.2

Filtering Model: We train a binary classifier based on a ROBERTA-BASE checkpoint on Spider’s train split to filter out inconsistent SQL-SQL pairs as described in Section 2.3.

Baselines: For baseline SQL-to-Text generation models, we consider recently proposed models like L2S (Wang et al., 2021), GAZP (Zhong et al., 2020), and SNOWBALL (Shu et al., 2021). All the baselines utilize pre-trained language models like BART (Lewis et al., 2020) or BERT (Devlin et al., 2018) for translating SQL tokens to natural text in a standard seq-to-seq set-up. The baselines mostly differ in the way of encoding SQL tokens as an input to the language model. In Section 3, we reviewed the recent SQL-to-Text methods.

Evaluation Metrics We evaluate the Text-to-SQL parsers using the Exact Set Match (EM), and the Exection Accuracy (EX) Yu et al. (2018). The EM metric measures set match for all the SQL clauses and returns 1 if there is a match across all the clauses. It ignores the DB-values (constants) in the SQL query. The EX metric directly compares the results obtained by executing the predicted query $\hat{q}$ and the gold query $q$ on the database.

We provide more implementation details including the hyperparameter settings in appendix A.5.

5 Results and Analysis

We first demonstrate the effectiveness of the synthetic data generated using R EfILL for fine-tuning Text-to-SQL parsers to new schemas. We compare with the recent methods that utilize SQL-to-Text generation for training-data augmentation (§ 5.1). We then evaluate the intrinsic quality of the synthetic data generated by different methods in terms of the text diversity and the agreement of the generated text with the ground truth (§ 5.2). We demonstrate that higher text diversity results in better performance of the adapted parsers (§ 5.3). We then justify the key design choices related to masking of the retrieved text and training of the schema translator module that improves the quality of R EfILL generated text (§ 5.4). Finally, we demonstrate the importance of using an independent binary classifier over cycle-consistency filtering (§ 5.5).

5.1 Evaluating adapted parsers

In Table 1, we compare the performance of parsers fine-tuned on Text-SQL pairs generated using R EfILL and other SQL-to-Text generation baselines. We observe that fine-tuning on high-quality and diverse text generated by R EfILL provides consistent performance gains over the base model across all the database groups. On average, R EfILL improves the base model by 8.0 EM in comparison to a gain of 2.8 EM by the best baseline (GAZP). We observe that the gains from baseline methods are

<table>
<thead>
<tr>
<th>Method</th>
<th>Group 1</th>
<th>Group 2</th>
<th>Group 3</th>
<th>Group 4</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE-M</td>
<td>80.9</td>
<td>84.3</td>
<td>64.8</td>
<td>67.2</td>
<td>64.0</td>
</tr>
<tr>
<td>L2S (Wang et al., 2021)</td>
<td><strong>88.7</strong></td>
<td><strong>87.8</strong></td>
<td>61.3</td>
<td>62.1</td>
<td>62.8</td>
</tr>
<tr>
<td>GAZP (Zhong et al., 2020)</td>
<td>85.2</td>
<td>85.2</td>
<td>58.9</td>
<td>66.9</td>
<td>70.1</td>
</tr>
<tr>
<td>SNOWBALL (Shu et al., 2021)</td>
<td>85.2</td>
<td><strong>87.8</strong></td>
<td>59.7</td>
<td>60.5</td>
<td>64.0</td>
</tr>
<tr>
<td>R EfILL (Ours)</td>
<td><strong>88.7</strong></td>
<td>87.0</td>
<td><strong>69.7</strong></td>
<td><strong>73.8</strong></td>
<td><strong>73.2</strong></td>
</tr>
</tbody>
</table>

Table 1: Results for finetuning a base semantic parser (SMBOP) on Text-SQL pairs generated using various SQL-to-Text baselines and R EfILL (§ 5.1). R EfILL provides consistent gains over the base model across all the database groups, while gains from other methods are often negative or small.
Figure 3: Average EM performance of Text-to-SQL models on the four groups vs. the size of SQL workload (§ 5.1). Data generated by REFill using 30% SQL workload yields better performance than the data from the existing best baseline utilizing 70% workload.

often small or even negative. REFill continues to yield positive gains even for smaller workload sizes — In Figure 3, we plot the fraction of the total SQL workload used on the x-axis and the EM of the fine-tuned parsers averaged across all the four groups, on the y-axis. When using the data synthesized by REFill, the performance of the parser improves steadily with an increasing size of the SQL workload. In contrast, the baseline SQL-to-Text generation methods fail to provide significant improvements. Interestingly, the data synthesized by REFill using the 30% SQL workload leads to better downstream performance of the adapted parsers than any of the baselines utilizing 70% SQL workload for SQL-to-Text generation.

5.2 Quality and Diversity of generated text

We explain our gains over existing methods due to the increased quality and diversity of the generated text. We measure quality using the BLEU score of the set \( S(q) \) of generated text for an SQL query \( q \), with the gold text of \( q \) as reference. To measure diversity we utilize SelfBLEU (Zhu et al., 2018) that measures the average BLEU score among text in \( S(q) \). Lower SelfBLEU implies higher diversity. We evaluate on all the gold SQL-Text pairs available in the Spider’s dev set. In Table 2, we compare the quality and diversity of the text generated using REFill with prior SQL-to-Text generation methods. For each method we generate 10 hypotheses per SQL query, and pick the hypothesis with the highest BLEU to report the overall BLEU scores. To allow baselines to generate more diverse text than the standard beam search, we utilize beam-sampling (Fan et al., 2018; Holtzman et al., 2019). For REFill, the 10 hypothesis come from using up to 10 retrieved-and-masked templates. We observe that our method of masking and refilling the natural text retrieved from existing datasets allows REFill to generate higher quality text (+8.4 BLEU) with naturally high text diversity.

5.3 Importance of Text Diversity

Retrieving and editing text from multiple existing examples enables REFill to generate diverse text. In Figure 4, we show that increased diversity of the generated text leads to improved performance of the fine-tuned parser. We vary the number of retrieved-and-masked templates on the x-axis and plot the performance of the fine-tuned parsers on the y-axis for each group. To maintain the number of synthesized examples the same, the product of beam-samples and the number of retrieved templates is held constant. We observe that fine-tuning
the parser on more diverse data generated using 5
retrieved templates per SQL provides consistently
superior EM performance across all the four groups
than using less diverse data obtained by retrieving
just one or two templates per SQL. The consistent
drops in EM while increasing the retrieved tem-
plates from 5 to 10 is explained by the reduction
in text diversity. Using 5 retrieved templates yields
a 100 – SelfBLEU score of 46.7, while with 10
retrieved templates 100 – SelfBLEU reduces to
33.8. This reduction is due to the inclusion of more
similar templates as we increase their number from
5 to 10. Finally, the drop in ReFill’s performance
with reduced text diversity reconfirms the worse
performance of SQL-to-Text baselines reported in
Section 5.1 that do not offer enough text diversity.

5.4 Design choices of Schema Translator
In Section 2.2, we described two important design
choices: (1) Method of masking schema-relevant
tokens and (2) Method of training the Edit-and-Fill
model for editing and refilling the masked text. We
justify these design choices by comparing the qual-
ity of the generated text with each combination of
these choices in Table 3. Comparing across rows
(Schema-Match Vs Frequency), we observe that
Frequency based masking results in 2 to 3 point
improvements in BLEU compared to masking by
matching schema names. Table A7 shows specific
examples where the schema-match method fails
to mask sufficiently. In contrast, even though the
frequency-based method might over-mask it still
suffices for our goal of guiding the text generation
model. Comparing across columns (Naïve Train
Vs. Robust Train) we observe that specifically train-
ing the template filling model for being robust to
the input templates also improves quality of the
generated text by 3.6 to 4.6 points.

5.5 Importance of Filtering model
Cycle-consistency based filtering (Zhong et al.,
2020) rejects a synthesized SQL-Text pair \((q, x)\)
if the output SQL \(\hat{q}\) generated by the base Text-to-
SQL parser for the input text \(x\) does not match with
the SQL \(q\). We argue that cycle-consistency based
filtering is sub-optimal for two reasons: (i) Data Re-
dundancy: Since the Text-to-SQL parser is already
capable of generating the correct output for the
retained examples, fine-tuning on these examples
does not offer much improvements. (ii) Data Loss:
If the base Text-to-SQL model is weak in parsing
text-queries for the target database, a large por-
tion of potentially useful training examples get get-
tered out due to cycle-inconsistency. In response,
we train a Filtering model as described in Sec-
tion 2.3. Since our filtering model is independent
of the base parser, it retains many useful generated
examples that might otherwise be filtered out by
cycle-consistency filtering using a weak Text-to-
SQL parser. In appendix Table A6, we provide in-
stances of useful training examples that get filtered
because of cycle-inconsistency, but are retained by
our filtering model. In Table 4, we compare the
base Text-to-SQL parser, with models fine-tuned
without any filtering, with cycle-consistent filter-
ing, and with using our filtering model. We focus
on Group-4 where the base Text-to-SQL parser is
significantly weaker compared to other groups, and
use ReFill to synthesize data for the 30% query
workload. Not using any filtering, or using cycle-
consistent filtering results in worse performance,
while applying our filtering model offers significant
improvements over the base model.

5.6 Experiments on datasets outside Spider
We further validate our method on four single-
database datasets outside Spider, namely Geo-
Query (Zelle and Mooney, 1996), Academic (Li
and Jagadish, 2014), IMDB and Yelp (Navid Yag-
ghmazadeh and Dillig, 2017). In Table 5, we first
report the performance of our base Text-to-SQL
parser and observe poor cross-database generaliza-
tion with an average EM of just 9.7. We adapt
Table 5: Evaluation on datasets outside Spider. We continue to observe that fine-tuning on data synthesized by ReFILL offers superior EM performance (§ 5.6).

<table>
<thead>
<tr>
<th>Method</th>
<th>Geo</th>
<th>Acad</th>
<th>IMDB</th>
<th>Yelp</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE-M</td>
<td>9.8</td>
<td>2.8</td>
<td>19.3</td>
<td>7.2</td>
<td>9.7</td>
</tr>
<tr>
<td>L2S</td>
<td>27.9</td>
<td>14.4</td>
<td>24.8</td>
<td>19.8</td>
<td>21.7</td>
</tr>
<tr>
<td>GAZP</td>
<td>20.8</td>
<td>16.0</td>
<td>19.3</td>
<td>10.8</td>
<td>16.7</td>
</tr>
<tr>
<td>SNOWBALL</td>
<td>25.6</td>
<td>8.8</td>
<td>21.1</td>
<td>18.9</td>
<td>18.6</td>
</tr>
<tr>
<td>ReFILL (Ours)</td>
<td><strong>30.1</strong></td>
<td><strong>27.6</strong></td>
<td><strong>26.6</strong></td>
<td><strong>29.7</strong></td>
<td><strong>28.5</strong></td>
</tr>
</tbody>
</table>

the base parser by fine-tuning it on the synthetic datasets generated by ReFILL and other baselines utilizing 30% of the available SQL workload. Table 5 compares the EM accuracy of the adapted parsers. We continue to observe that fine-tuning on datasets synthesized by ReFILL consistently outperforms prior SQL-to-Text based data-augmentation baselines. In appendix Table A4, we provide results for 50% and 70% workload settings.

6 Conclusion

We presented ReFILL, a framework for generating diverse and high quality parallel data for adapting existing Text-to-SQL parsers to a target database. ReFILL translates a given SQL workload into a dataset of Text-SQL pairs containing diverse text queries. To achieve higher diversity, ReFILL retrieves and edits examples from existing datasets and transfers them to a target schema. Through extensive experiments across multiple databases, we demonstrate that fine-tuning Text-to-SQL parsers on datasets generated using ReFILL result in more accurate adaptation to target schemas. Even with smaller SQL workloads ReFILL often outperforms the SQL-to-Text generation baselines utilizing larger SQL workloads.

7 Limitations

This work focuses on synthesizing parallel data containing diverse text queries for adapting pre-trained Text-to-SQL models to new databases. Thus, our current effort toward diverse text query generation using ReFILL is limited to the Text-to-SQL semantic parsing task. Extending ReFILL for data-augmentation in other semantic parsing or question-answering tasks is an exciting direction we hope to explore as part of future work.

Our experimental set-up assumes a small workload of real SQL queries. As per Baik et al. (2019), a small workload of real SQL queries is a reasonable assumption since SQL query logs are often available for existing in-production databases that are to be supported by a Text-to-SQL service. Synthesizing realistic SQL-query workloads for newly instantiated databases is a challenging and promising direction but different from the diverse text-query generation aspect of our work.

8 Ethical Considerations

Our goal with ReFILL is to synthesize parallel data for adapting Text-to-SQL parsers to new schemas. We believe that the real-world deployment of Text-to-SQL or any semantic parser trained on text generated by language models must go through a careful review of any harmful biases. Also, the intended users of any Text-to-SQL service must be made aware that the answers generated by these systems are likely to be incorrect. We do not immediately foresee any serious negative implications of the contributions that we make through this work.

9 Acknowledgements

We thank the reviewers from ACL rolling review for their insightful and quality feedback. We gratefully acknowledge support from IBM Research, specifically the IBM AI Horizon Networks-IIT Bombay initiative. Abhijeet thanks Google for supporting his research with Google PhD Fellowship.

References

Christopher Baik, H. V. Jagadish, and Yunyao Li. 2019. Bridging the semantic gap with SQL query logs in natural language interfaces to databases. In 35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019, pages 374–385. IEEE.


generation meets scaffolded, structured dialogue. *Alexa Prize Proceedings.*


Juncen Li, Robin Jia, He He, and Percy Liang. 2018. Delete, retrieve, generate: a simple approach to sentiment and style transfer. In *NAACL-HLT.*


A Appendix

A.1 Dataset Details

<table>
<thead>
<tr>
<th>Group</th>
<th>Number of queries by hardness</th>
<th>Number of examples</th>
<th>Unique SQLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>easy medium hard extra</td>
<td>115</td>
<td>62</td>
</tr>
<tr>
<td>• concert_singer</td>
<td>4 24 13 4</td>
<td>45</td>
<td>26</td>
</tr>
<tr>
<td>• singer</td>
<td>6 18 6 0</td>
<td>30</td>
<td>16</td>
</tr>
<tr>
<td>• orchestra</td>
<td>14 18 6 2</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>Group 2</td>
<td>easy medium hard extra</td>
<td>124</td>
<td>65</td>
</tr>
<tr>
<td>• dog_kennels</td>
<td>10 36 10 26</td>
<td>82</td>
<td>42</td>
</tr>
<tr>
<td>• pets_1</td>
<td>4 22 6 10</td>
<td>42</td>
<td>23</td>
</tr>
<tr>
<td>Group 3</td>
<td>easy medium hard extra</td>
<td>164</td>
<td>84</td>
</tr>
<tr>
<td>• students_transcripts_tracking</td>
<td>26 24 8 20</td>
<td>78</td>
<td>41</td>
</tr>
<tr>
<td>• course_teach</td>
<td>8 14 8 0</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td>• network_1</td>
<td>12 22 16 6</td>
<td>56</td>
<td>28</td>
</tr>
<tr>
<td>Group 4</td>
<td>easy medium hard extra</td>
<td>120</td>
<td>65</td>
</tr>
<tr>
<td>• world_1</td>
<td>24 46 20 60</td>
<td>120</td>
<td>65</td>
</tr>
</tbody>
</table>

Table A1: Number of schemas and statistics of query workload for each group. Related schemas were grouped together in order to obtain larger evaluation sets per group.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of queries by hardness</th>
<th>Number of examples</th>
<th>Unique SQLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geoquery (Geo)</td>
<td>easy medium hard extra</td>
<td>553</td>
<td>210</td>
</tr>
<tr>
<td>Academic (Acad)</td>
<td>224 32 220 77</td>
<td>181</td>
<td>176</td>
</tr>
<tr>
<td>IMDB</td>
<td>23 12 48 26</td>
<td>109</td>
<td>75</td>
</tr>
<tr>
<td>Yelp</td>
<td>13 29 25 24</td>
<td>111</td>
<td>98</td>
</tr>
</tbody>
</table>

Table A2: Statistics of queries in additional (non-spider) datasets. We utilize the pre-processed versions of these datasets provided by Yu et al. (2018).

A.2 Results in low or medium SQL workload setting

<table>
<thead>
<tr>
<th>Group 1</th>
<th>Group 2</th>
<th>Group 3</th>
<th>Group 4</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Method</strong></td>
<td><strong>EM</strong></td>
<td><strong>EX</strong></td>
<td><strong>EM</strong></td>
<td><strong>EX</strong></td>
</tr>
<tr>
<td><strong>Results with 30% SQL workload</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BASE-M</td>
<td>80.9</td>
<td>84.3</td>
<td>64.8</td>
<td>67.2</td>
</tr>
<tr>
<td>L2S</td>
<td>82.6</td>
<td>84.3</td>
<td>60.5</td>
<td>65.3</td>
</tr>
<tr>
<td>GAZP</td>
<td>83.5</td>
<td>84.3</td>
<td>61.3</td>
<td>64.5</td>
</tr>
<tr>
<td>SNOWBALL</td>
<td>80.0</td>
<td>83.5</td>
<td>59.7</td>
<td>63.7</td>
</tr>
<tr>
<td>REFILL</td>
<td><strong>86.1</strong></td>
<td><strong>86.1</strong></td>
<td><strong>65.6</strong></td>
<td><strong>65.6</strong></td>
</tr>
<tr>
<td><strong>Results with 50% SQL workload</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BASE-M</td>
<td>80.9</td>
<td>84.3</td>
<td>64.8</td>
<td>67.2</td>
</tr>
<tr>
<td>L2S</td>
<td>89.6</td>
<td>88.7</td>
<td>66.1</td>
<td>68.5</td>
</tr>
<tr>
<td>GAZP</td>
<td>87.8</td>
<td>87.0</td>
<td>58.9</td>
<td>63.7</td>
</tr>
<tr>
<td>SNOWBALL</td>
<td>83.5</td>
<td>85.2</td>
<td>55.6</td>
<td>66.1</td>
</tr>
<tr>
<td>REFILL</td>
<td><strong>88.7</strong></td>
<td><strong>91.3</strong></td>
<td><strong>67.2</strong></td>
<td><strong>69.7</strong></td>
</tr>
<tr>
<td><strong>Results with 70% SQL workload</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BASE-M</td>
<td>80.9</td>
<td>84.3</td>
<td>64.8</td>
<td>67.2</td>
</tr>
<tr>
<td>L2S</td>
<td><strong>88.7</strong></td>
<td><strong>87.8</strong></td>
<td>61.3</td>
<td>62.1</td>
</tr>
<tr>
<td>GAZP</td>
<td>85.2</td>
<td>85.2</td>
<td>58.9</td>
<td>66.9</td>
</tr>
<tr>
<td>SNOWBALL</td>
<td>85.2</td>
<td><strong>87.8</strong></td>
<td>59.7</td>
<td>60.5</td>
</tr>
<tr>
<td>REFILL</td>
<td><strong>88.7</strong></td>
<td>87.0</td>
<td><strong>69.7</strong></td>
<td><strong>73.8</strong></td>
</tr>
</tbody>
</table>

Table A3: Evaluation on four groups of schemas held out from Spider’s dev set, for varying sizes of query workload [30%, 50%, 70%] used for SQL-to-Text translation.
Table A4: EM evaluation on four additional datasets outside Spider, for varying sizes of query workload {30%, 50%, 70%} used for SQL-to-Text translation. Since the contents of Acad, IMDB, and Yelp databases were not publicly accessible to us, we are unable to report EX results on these databases. EX results for GeoQuery appear in Table A5.

Table A5: EX accuracy evaluation on GeoQuery dataset, for varying sizes of query workload {30%, 50%, 70%}.

A.3 Examples rejected by cycle-consistency but retained by our filtering model

<table>
<thead>
<tr>
<th>Generated text</th>
<th>Gold SQL</th>
<th>Predicted SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>How many countries are governed by Islamic Emirate?</td>
<td>SELECT count(*) FROM country WHERE GovernmentForm = 'Islamic Emirate'</td>
<td>SELECT COUNT(*) FROM country WHERE country.code NOT IN (SELECT countrylanguage.countrycode FROM countrylanguage)</td>
</tr>
<tr>
<td>What is the number of languages that are official in Australia?</td>
<td>SELECT COUNT(*) FROM country AS T1 JOIN countrylanguage AS T2 ON T1.Code = T2.CountryCode WHERE T1.Name = 'Australia' AND IsOfficial = 'T'</td>
<td>SELECT COUNT(*) FROM countrylanguage JOIN country ON countrylanguage.countrycode = country.code WHERE country.name = 'Australia'</td>
</tr>
<tr>
<td>Find the language of the country that has the head of state Salahuddin Abdul Aziz Shah Alhaj and is official.</td>
<td>SELECT T2.Language FROM country AS T1 JOIN countrylanguage AS T2 ON T1.Code = T2.CountryCode WHERE T1.HeadOfState = 'Salahuddin Abdul Aziz Shah Alhaj' AND T2.IsOfficial = 'T'</td>
<td>SELECT countrylanguage.language FROM countrylanguage JOIN country ON countrylanguage.countrycode = country.code WHERE country.headofstate = 'Salahuddin Abdul Aziz Shah Alhaj'</td>
</tr>
<tr>
<td>What are the names of countries with surface area greater than the smallest area of any country in Antarctica?</td>
<td>SELECT Name FROM country WHERE SurfaceArea &gt; (SELECT min(SurfaceArea) FROM country WHERE Continent = 'Antarctica')</td>
<td>SELECT country.name FROM country WHERE country.surfacearea &gt; (SELECT MAX(country.surfacearea) FROM country WHERE country.continent = 'Antarctica')</td>
</tr>
</tbody>
</table>

Table A6: Consistent SQL-Text pairs rejected by cycle-consistency but retained by our filtering model. Predicted SQL is the output of the Text-to-SQL model used for checking cycle consistency, and does not match the gold SQL often due to minor errors.
A.4 Examples of masking

### SQL

```sql
SELECT T1.template_type_code, count(*) FROM Templates AS T1 JOIN Documents AS T2 ON T1.template_id = T2.template_id GROUP BY T1.template_type_code  
[Schema Name: Document Template Management]
```

**Reference**

Show all template type codes and the number of documents using each type.

**Retrieved SQL**

```sql
T1.FacID, count(*) FROM Faculty AS T1 JOIN Student AS T2 ON T1.FacID = T2.advisor GROUP BY T1.FacID  
[Schema Name: Faculty Student Activity]
```

**Retrieved Text**

Show the faculty id of each faculty member, along with the number of students he or she advises.

**Sch-match Mask**

Show the **MASK** of each **MASK** member, along with the number of **MASK** he or she **advises**.

**Freq Mask**

Show the **MASK** of each **MASK**, **MASK** with the number of **MASK** he or she **MASK**.

**Filled Text**

Show the **type code** of each **template** **member**, along with the number of **documents** he or she **advises**.

### Filled Text

Show the type code of each template member, along with the number of documents he or she advises.

### SQL

```sql
SELECT T2.name, T2.capacity FROM concert AS T1 JOIN stadium AS T2 ON T1.stadium_id = T2.stadium_id WHERE T1.year >= 2014 GROUP BY T2.stadium_id ORDER BY count(*) DESC LIMIT 1  
[Schema Name: Concert Singer]
```

**Reference**

Show the stadium name and capacity with most number of concerts in year 2014 or after.

**Retrieved SQL**

```sql
SELECT T2.name, T1.team_id_winner FROM postseason AS T1 JOIN team AS T2 ON T1.team_id_winner = T2.team_id_br WHERE T1.year = 2008 GROUP BY T1.team_id_winner ORDER BY count(*) DESC LIMIT 1  
[Schema Name: Baseball 1]
```

**Retrieved Text**

What are the name and id of the team with the most victories in 2008 postseason?

**Sch-match Mask**

What are the **MASK** and **MASK** of the **MASK** with the most **victories** in **MASK**

**Freq Mask**

What are the **MASK** and **MASK** of the **MASK** with the most **MASK** in **MASK**

**Filled Text**

What are the name and capacity of the stadium with the most victories in year 2014?

What are the name and capacity of the stadium with the most concerts in 2014?

---

Table A7: Masking the text based on string matches Vs. our method of frequency based masking. Schema-relevant words like ‘victories’, ‘members’, ‘advises’ that do not have a sufficient string match with any of the table or column names of their schema, get left out when using string-match based matches. Thus failing to mask the words in the original schema might lead to copying of the word in the target schema, thus making the generated text semantically inconsistent. Words in **blue** are schema relevant words for the target database and should appear in the generated output.
A.5 Hyperparameters

Our Edit and Fill model (139.2M parameters) is based on a pretrained BART-BASE (Lewis et al., 2020) model. We fine-tune this model for 100 epochs with learning rate of $3 \times 10^{-5}$, weight decay of 0.01 and batch size of 64. The pretrained model is obtained from HuggingFace\(^3\).

The proposed binary classifier (124.6M params) is pretrained RoBERTA-BASE (Liu et al., 2020) (obtained from HuggingFace\(^4\)) fine-tuned for 100 epochs on our data with learning rate $10^{-5}$, weight decay 0.01 and batch size 16 for 100 epochs.

For SMBoP experiments, we use a smaller SMBoP model with 4 RAT layers and RoBERTA-BASE (Liu et al., 2020) encoder as a baseline. The number of parameters in this model is 132.9M. All the adaptation experiments use learning rate of $5 \times 10^{-6}$, learning rate of language model of $3 \times 10^{-6}$ and batch size of 8. All the models were trained for 100 epochs.

All the experiments were performed on a single NVIDIA RTX 3060 GPU. Training times for the Edit-and-Fill model and binary classifiers were $\approx 4.5$ hrs and $\approx 6.5$ hrs respectively. Each of the fine-tuning experiment took $3 \sim 4$ hrs to complete.

A.6 Cost function for Tree Edit Distance

<table>
<thead>
<tr>
<th>Group</th>
<th>Value</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td>Equal</td>
<td>0</td>
</tr>
<tr>
<td>Equal</td>
<td>Unequal</td>
<td>0.5</td>
</tr>
<tr>
<td>Unequal</td>
<td>Equal</td>
<td>0</td>
</tr>
<tr>
<td>Unequal</td>
<td>Unequal</td>
<td>1</td>
</tr>
</tbody>
</table>

Table A8: Cost function of nodes $n_1$ and $n_2$ based on their groups and value.

A.7 Examples of TED neighbours

```
(a) SELECT count(*) FROM head WHERE age > 56
Table(SELECT) Agg(COUNT) Table(FROM) Value(*) Predicate(>) Table(head)
Value(age) Value(56)

(b) SELECT count(*) FROM game WHERE season > 2007
Table(SELECT) Agg(COUNT) Table(FROM) Value(*) Predicate(>) Table(game)
Value(season) Value(2007)
```

Figure 5: Example of tree pair with TED=0

```
(a) SELECT count(*) FROM county_public_safety
Table(SELECT) Agg(AVG) Table(film)
Value(*) Value(Gross_in_dollar)
```

Figure 6: Example of tree pair with non-zero(=0.125) TED

We use APTED library (Pawlik and Augsten, 2015, 2016) to compute TED between 2 parsed SQL trees. For every node in the tree, a group is assigned according to table A9. Then the cost for various combinations of node groups and node values is described in table A8. If either of the nodes does not belong to any of the groups in table A9, their groups are considered to be “unequal” and cost will be assigned based on their values.