Semantic Evaluation for Text-to-SQL with Distilled Test Suites

Ruiqi Zhong[*] **Tao Yu**[†] **Dan Klein**[*]
[*] Computer Science Division, University of California, Berkeley
[†] Department of Computer Science, Yale University
ruiqi-zhong@berkeley.edu tao.yu@yale.edu klein@berkeley.edu

Abstract

We propose *test suite accuracy* to approximate semantic accuracy for Text-to-SQL models. Our method distills a small test suite of databases that achieves high code coverage for the gold query from a large number of randomly generated databases. At evaluation time, it computes the denotation accuracy of the predicted queries on the distilled test suite, hence calculating a tight upper-bound for semantic accuracy efficiently. We use our proposed method to evaluate 21 models submitted to the SPIDER leader board and manually verify that our method is always correct on 100 examples. In contrast, the current SPIDER metric leads to a 2.5% false negative rate on average and 8.1% in the worst case, indicating that test suite accuracy is needed. Our implementation, along with distilled test suites for eleven Text-to-SQL datasets, is publicly available.[1]

1 Introduction

A Text-to-SQL model translates natural language instructions to SQL queries that can be executed on databases and bridges the gap between expert programmers and non-experts. Accordingly, researchers have built a diversity of datasets (Dahl, 1989; Iyer et al., 2017; Zhong et al., 2017; Yu et al., 2018) and improved model performances (Xu et al., 2017; Suhr et al., 2018; Guo et al., 2019; Bogin et al., 2019a; Wang et al., 2020). However, evaluating the semantic accuracy of a Text-to-SQL model is a long-standing problem: we want to know whether the predicted SQL query has the same denotation as the gold for every possible database. "Single" denotation evaluation executes the predicted SQL query on one database and compares its denotation with that of the gold. It might create *false positives*, where a semantically different prediction (Figure 1 prediction 1) happens to have the same denotation as the gold, on a particular database. In contrast, exact string match might produce *false negatives*: Figure 1 prediction 2 is semantically equivalent to the gold but differs in logical form.

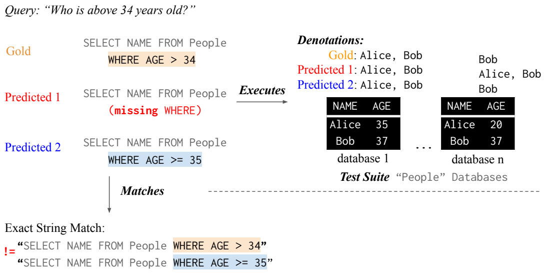


Figure 1: Prediction 2 is semantically correct, and Prediction 1 is wrong. Exact string match judges prediction 2 to be wrong, which leads to *false negatives*. Only comparing denotations on database 1 judges prediction 1 to be correct, which leads to *false positives*. Test suite evaluation compares denotations on a set of databases and reduces false positives.

The programming language research community developed formal tools to reliably reason about query equivalence for a restricted set of query types. They lift SQL queries into other semantic representations such as K-relations (Green et al., 2007), UniNomial (Chu et al., 2017) and U-semiring (Chu et al., 2018); then they search for an (in)equivalence proof. However, these representations cannot express sort operations and float comparisons, and hence do not support the full range of operations that Text-to-SQL models can use. We ideally need a method to approximate semantic accuracy reliably without operation constraints.

If the computational resources were unlimited, we could compare the denotations of the predicted query with those of the gold on a large number of random databases (Section 4.1), and obtain a tighter upper bound for semantic accuracy than single de-

[1] Metric implementation and test suites available here, for datasets: SPIDER, CoSQL, SPARC, Academic, Advising, ATIS, Geography, IMDB, Restaurants, Scholar and Yelp.

notation evaluation. The software testing literature calls this idea fuzzing (Padhye et al., 2019; AFL; Lemieux et al., 2018; Qui). However, it is undesirable to spend a lot of computational resources every time when we evaluate a Text-to-SQL model. Instead, we want to check denotation correctness only on a smaller test suite of databases that are more likely to distinguish[2] *any* wrong model-predicted queries from the gold.

We propose *test suite accuracy* (Section 2) to efficiently approximate the semantic accuracy of a Text-to-SQL model, by checking denotations of the predicted queries on a compact test suite of databases with high code coverage. We introduce how to construct/search for such a test suite without prior information about model-predicted queries.

Our search objective is formally defined through *neighbor queries* (Section 3.1), which are generated by modifying one aspect of the gold query. For example, prediction 1 in Figure 1 is a neighbor query of the gold, since they differ only by a "`WHERE`" clause. These neighbor queries are usually semantically different from the gold, and if a test suite can distinguish them from the gold, it is likely to distinguish other wrong queries as well. The latter holds because distinguishing all neighbors from the gold requires executions on these databases to exercise every modified part of the gold query, hence reflecting comprehensive code coverage and high test quality (Miller and Maloney, 1963; Ammann and Offutt). Hence, we formalize our objective as finding a small test suite that can distinguish all the neighbors (Section 3.2).

We search under this objective by generating a large number of random databases (Section 4.1) and keeping a small fraction of them that can distinguish the neighbors from the gold (Section 4.2). We call this set of databases a *distilled test suite*. While evaluating model-predicted queries, we only check their denotations on the distilled test suite to approximate semantic accuracy efficiently.

Application We distill a test suite for the SPIDER dataset (Yu et al., 2018) (Section 5) from 1000 random databases, which can distinguish more than 99% of the neighbor queries. We use the test suite to evaluate 21 SPIDER leader board submissions, randomly sample 100 model-predicted queries where our method disagrees with exact set match (ESM, the current SPIDER official metric), and manually verify that our method is correct in *all* these cases (Section 6.1).

We use test suite accuracy as a proxy for semantic accuracy to examine how well ESM approximates the semantic accuracy (Section 6.2), and identify several concerns. (1) ESM tends to underestimate model performances, leading to a 2.5% false negative rate on average and 8.1% in the worst case. (2) ESM does not reflect all improvements in semantic accuracy. For example, it undervalues a high-score submission with 61% semantic accuracy by 8%, but instead favors five other submissions with lower semantic accuracy, thus misrepresenting state of the art. (3) ESM becomes poorer at approximating semantic accuracy on more complex queries. Since models are improving and producing harder queries, ESM deviates more from semantic accuracy. We need test suite accuracy to better track progress in Text-to-SQL development.

Our main paper focuses on SPIDER. However, we also generated distilled test suites for other popular text-to-SQL datasets including CoSQL (Yu et al., 2019a), SPARC (Yu et al., 2019b), Academic (Li and Jagadish, 2014), Advising (Finegan-Dollak et al., 2018a), ATIS (Dahl et al., 1994), Geography (Zelle and Mooney, 1996), IMDB (Yaghmazadeh et al., 2017), Restaurants (Popescu et al., 2003), Scholar (Iyer et al., 2017) and Yelp (Yaghmazadeh et al., 2017). We will release our test suites[3] and the details of these datasets can be seen in Appendix A.2.

To summarize, we contribute:

- A method and a software to create compact high quality test suites for Text-to-SQL semantic evaluation.
- Test suites to reliably approximate semantic accuracy for eleven popular datasets.
- A detailed analysis of why current metrics are poor at approximating semantic accuracy.

2 Problem Statement

Let $w \in \mathcal{W}$ be a database input to a SQL query $q \in \mathcal{Q}$, and $[\![q]\!]_w$ be the denotation of q on w,[4] where $\mathcal{W}/\mathcal{Q}$ is the space of all databases/SQL queries. Two queries q_1 and q_2 are semantically equivalent

[2] Section 2 defines that a database distinguishes two queries if their executions lead to different results.

[3] https://github.com/ruiqi-zhong/TestSuiteEval

[4] As in Yu et al. (2018), we use Sqlite3 to obtain the denotation. Define $[\![q]\!]_w = \bot$ if execution does not end, which is implemented as timeout in practice.

if their denotations are the same for all possible databases, i.e.

$$\forall w \in \mathcal{W}, [\![q_1]\!]_w = [\![q_2]\!]_w \tag{1}$$

We refer to the ground truth query as g and the model-predicted query to be evaluated as q. Ideally, we want to evaluate whether q is semantically equivalent to g (abbreviated as **semantic accuracy**), which is unfortunately undecidable in general (Chu et al., 2017). Traditionally, people evaluate a model-predicted query q by either **exact string match** or compare denotations on a single database w (abbreviated as **single denotation accuracy**). Exact string match is too strict, as two different strings can have the same semantics. Single denotation evaluation is too loose, as the denotations of g and q might be different on another database w.

We use **test suite** to refer to a set of databases. A database w **distinguishes** two SQL queries g, q if $[\![g]\!]_w \neq [\![q]\!]_w$, and a test suite S distinguishes them if one of the databases $w \in S$ distinguishes them:

$$\exists w \in S, [\![g]\!]_w \neq [\![q]\!]_w \tag{2}$$

For convenience, we define the indicator function:

$$D_S(g, q) := \mathbb{1}[S \text{ distinguishes } g, q] \tag{3}$$

We use the test suite S to evaluate a model-predicted query q: q is correct iff $D_S(g, q) = 0$; i.e., g and q have the same denotations on all databases in S.

To summarize, if $M_1 \Rightarrow M_2$ means that "correctness under M_1 implies correctness under M_2", exact match ⇒ semantic accuracy ⇒ test suite accuracy ⇒ single denotation accuracy. Our goal is to construct a test suite of databases S to obtain a tight upper bound on semantic accuracy with test suite accuracy reliably and efficiently.

3 Desiderata

Since we want to construct a test suite S of databases for each gold query g, we use S_g to denote the target test suite. Before describing how to generate S_g, we first list two criteria of a desirable test suite. Later we construct S_g by optimizing over these two criteria.

Computational Efficiency. We minimize the size of S_g to speed up test suite evaluations.

Gold *g*	SELECT NAME FROM People
	WHERE AGE >= 34 AND NAME LIKE "%Alice%"

Replace Column Name	SELECT AGE FROM People
	WHERE AGE >= 34 AND NAME LIKE "%Alice%"

Replace Comparison	SELECT NAME FROM People
	WHERE AGE > 34 AND NAME LIKE "%Alice%"

Replace Numerical	SELECT NAME FROM People
	WHERE AGE >= 33 AND NAME LIKE "%Alice%"

Replace String	SELECT NAME FROM People
	WHERE AGE >= 34 AND NAME LIKE "%Bob%"

Drop Span	SELECT NAME FROM People
	WHERE ~~AGE >= 34 AND~~ NAME LIKE "%Alice%"

Figure 2: Automatically generating a set of neighbor queries N_g. We apply one type of modification to the original gold query g at a time. The modified queries are likely to be semantically close but inequivalent to the gold.

Code Coverage. The test suite needs to cover every branch and clause of the gold query such that it can test the use of every crucial clause, variable, and constant. For example, database 1 in Figure 1 alone does not have a row where "AGE ≤ 34" and hence does not have comprehensive code coverage.

3.1 Measure Coverage through Neighbors

We measure the code coverage of a test suite by its ability to distinguish the gold query from its *neighbor queries*: a set of SQL queries that are close to the gold in surface forms but likely to be semantically different. To generate them, we modify one of the following aspects of the gold query (Figure 2): (1) replace an integer (float) constant with either a random integer (float) or its value ± 1 (0.001); (2) replace a string with a random string, its sub-string or a concatenation of it with another random string; (3) replace a comparison operator/column name with another; (4) drop a query span unless the span does not change the semantics of the query. For example, the "ASC" keyword does not change the semantics because it is the default sort order. We then remove modified queries that cannot execute without any errors.

Note that our method does not pre-determine the number of neighbor queries. It is adaptive and generates more neighbors for longer and more complex queries since there are more spans to drop and more constants to replace.

Neighbor queries have two desirable properties. First, they are likely to be semantically different from the gold query. For example, "> 34" is semantically different from "≥ 34" (replace comparison operator) and "> 35" (replace constants);

however, we only apply one modification at a time, since “$>$ 34” is semantically equivalent to “$\geq$ 35” for an integer. Secondly, their subtle differences from the gold require the test suite to cover all the branches of the gold query. For example, the database needs to have people above, below and equal to age 34 to distinguish all its neighbors. Hence, the test suite tends to have high quality if it can distinguish the gold from all its neighbors.

We use N_g to denote the set of neighbor queries of the gold query g.

3.2 Optimization Objective

To recap, our objective is to search for a *small* test suite S_g that can distinguish as *many* neighbor queries as possible. Formally, we optimize over S_g with the objective below:

$$\begin{aligned} & \text{minimize } |S_g| \\ s.t. \quad & \forall q \in N_g, D_{S_g(g,q)} = 1 \end{aligned} \tag{4}$$

4 Fuzzing

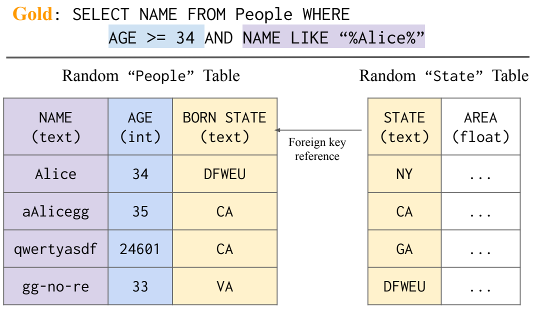
We optimize the above objective through fuzzing: a software testing technique that generates a large number of random inputs to test whether a program satisfies the target property (e.g., SQL equivalence). We describe a procedure to sample a large number of random databases and keep a small fraction of them to distill a test suite S_g.

4.1 Sampling Databases

A database w needs to satisfy the input type constraints of the gold program g, which include using specific table/column names, foreign key reference structure, and column data types. We describe how to generate a random database under these constraints and illustrate it with Figure 3.

If a column c_1 refers to another column c_2 as its foreign key, all elements in c_1 must be in c_2 and we have to generate c_2 first. We define a partial order among the tables: table A < table B if B has a foreign key referring to any column in table A. We then generate the content for each table in ascending order found by topological sort. For example, in Figure 3, we generate the “State” table before the “People” table because the latter refers to the former.

We now sample elements for each column such that they satisfy the type and foreign key constraints. If a column c_1 is referring to another column c_2, each element in c_1 is uniformly sampled from c_2. Otherwise, if the column is a numerical(string) type, each element is sampled uniformly from $[-2^{63}, 2^{63}]$ (a random string distribution). We also randomly add in constant values used in g (e.g., 34 and “Alice”) and their close variants (e.g., 35 and “aAlicegg”) to potentially increase code coverage. We denote the database distribution generated by this procedure as $\mathcal{I}_g$.



Gold: SELECT NAME FROM People WHERE AGE >= 34 AND NAME LIKE “%Alice%”

Random “People” Table

NAME (text)	AGE (int)	BORN STATE (text)
Alice	34	DFWEU
aAlicegg	35	CA
qwertyasdf	24601	CA
gg-no-re	33	VA

Random “State” Table

STATE (text)	AREA (float)
NY	...
CA	...
GA	...
DFWEU	...

Figure 3: A random database input w from the distribution $\mathcal{I}_g$, where g is the gold SQL query. We generate the “State” column before the “BORN STATE” column because the latter has to be a subset of the former. Each element of the column “BORN STATE” is sampled uniformly at random from the parent “STATE” column. For the column that has data type int/string, each element is either a random number/string or a close variant of a constant used the gold query.

4.2 Distilling a Test Suite

We use samples from $\mathcal{I}_g$ to construct a small test suite S_g such that it can distinguish as many neighbor queries (Section 3.1) in N_g as possible. We initialize S_g to be empty and proceed greedily. A database w is sampled from the distribution $\mathcal{I}_g$; if w can distinguish a neighbor query that cannot be distinguished by any databases in S_g, we add w to S_g. Appendix A.1 gives a more rigorous description. In the actual implementation, we also save the disk space by sharing the same random database w_t across all gold SQL queries that are associated with the same schema. Though this algorithm is far from finding the optimal solution to Objective 4, in practice, we find a test suite that is small enough to distinguish most neighbor queries.

5 Evaluation Setup

We introduce the dataset and the model-predicted queries we use to study our test suite evaluation. We also adapt our test suite evaluation and the official Spider metric to make a fair comparison.

5.1 Dataset

We generate test suites S_g for the development set of SPIDER (Yu et al., 2018), which contains 1034 English utterances and their corresponding SQL queries, spanning across 20 different database schemata. It stratifies data into four categories (easy, medium, hard, and extrahard) according to difficulty level measured by gold SQL complexity. We decide to focus on SPIDER because it invites researchers to submit their model-predicted queries and requires them to follow a standard format, which makes it convenient to study a wide variety of model-predicted queries.

5.2 Model-Predicted Queries

We obtain the development set model-predicted queries from 21 submissions.[5] They include models from Guo et al. (2019); Bogin et al. (2019b); Choi et al. (2020); Wang et al. (2020). [6] These models capture a broad diversity of network architectures, decoding strategies, and pre-traning methods, with accuracy ranging from below 40% to above 70%. The first author obtained these model-predicted queries from the second author *after* producing the test suites to ensure that our method is general and not tailored to a specific family of model-predicted queries.

5.3 Metric Adaptation

The SPIDER official evaluation metric is exact set match (abbreviated as ESM) (Zhong et al., 2017; Yu et al., 2018). It parses the gold and model-predicted queries into sub-clauses and determines accuracy by checking whether they have the same *set* of clauses. It improves over exact string matching by preventing false negatives due to semantically equivalent clause reordering. However, it is still considered a strict metric and creates false negatives.

To further reduce false negatives, the actual implementation of the official SPIDER metric is looser. We list all of its major differences from the standard ESM below; accordingly, we either adapt our test suite evaluation or fix the SPIDER implementation to make a fair comparison.

(1) The SPIDER metric does not check constant prediction correctness. Therefore, our adapted test suite evaluation enumerates all possible ways to replace the constants in a model-predicted query with the gold constants and consider a model-predicted query to be correct if one of the replacements passes the test suite. (2) The SPIDER metric does not check column order, so our adapted evaluation considers two denotations equivalent if they only differ by column order. (3) The SPIDER evaluation script accidentally ignores any join predicate. We fix this bug. (4) The SPIDER metric does not check table variable names. Yu et al. (2018) implemented this because different intermediate tables can contain the same column, hence selecting any of them is equivalent. We keep this feature since it effectively rules out many false negatives. However, it also introduces new false positives (e.g., Figure 8 row 1).

Unless we explicitly specify, in the rest of our paper, "ESM" and "test suite accuracy" refer to these adapted metrics rather than the original ones.

6 Results

We first show that our test suite evaluation is reliable by verifying that the test suite distinguishes most neighbor queries, and always makes the correct judgement on 100 model-predicted queries we manually examined (Section 6.1). Then we use test suite accuracy as a proxy for semantic accuracy to calculate the error rate of the existing commonly used metrics (Section 6.2) and their correlations (Section 6.3). At last we discuss the computational efficiency of test suite evaluation (Section 6.4).

6.1 Reliability

Distinguish Neighbor Queries. For each gold query in SPIDER, we generate on average 94 neighbor queries (Figure 2). We sample 1000 random databases for each database schema and run fuzzing (Section 4.2) to construct S_g, which takes around a week on 16 CPUs. Figure 5 plots the fraction of neighbor queries that remain undistinguished after attempting t random databases.

Checking single database denotation fails to distinguish 5% of the neighbor queries, and the curve stops decreasing after around 600 random databases; 1000 random databases can distinguish > 99% of the neighbor queries. A large number of random databases is necessary to achieve comprehensive code coverage.

Figure 4 presents some typical neighbor queries that have the same denotations as the gold on all the databases we sampled. These queries are only a small fraction (1%) of all the neighbors; in most

[5] The model-predicted queries are available here.

[6] Many dev set submissions do not have public references.

Modification	Gold & Modified	Passing Reason

Comparison Operator Replaced	`Gold: SELECT T1.NAME FROM Conductor …`
	`GROUP BY T2.CONDUCTER_ID HAVING COUNT(*) > 1`
	`Modified: SELECT T1.NAME FROM Conductor …`
	`GROUP BY T2.CONDUCTER_ID HAVING COUNT(*) != 1`
		Count is always positive, so "> 1" is equivalent to "!= 1". modification is semantically equivalent to the original SQL.

Constant Replaced	`Gold: SELECT NAME FROM City`
	`WHERE POPULATION BETWEEN 160000 AND 90000`
	`Modified: SELECT NAME FROM City`
	`WHERE POPULATION BETWEEN 160000 AND 21687`
		Original annotation is wrong and both the original and the modification lead to empty results, which are semantically equivalent.

Column Name Dropped	`Gold: SELECT COUNT(T2.LANGUAGE), T1.NAME …`
	`Modified: SELECT COUNT(*), T1.NAME FROM …`
		The SQL interpreter infers it should count the number of rows. modification is semantically equivalent to the original SQL.

Comparison Operator Replaced	`Gold: SELECT COUNT(*) FROM Dogs`
	`WHERE age < (SELECT AVG(AGE) FROM Dogs)`
	`Modified: SELECT COUNT(*) FROM Dogs`
	`WHERE age <= (SELECT AVG(AGE) FROM Dogs)`
		A dog entry needs to have exactly the average age to distinguish the modification. This happens with low probability and our test suite fails to distinguish them.

Figure 4: Representative modifications in N_g that produce the same results as the gold (pass) on all sampled databases.

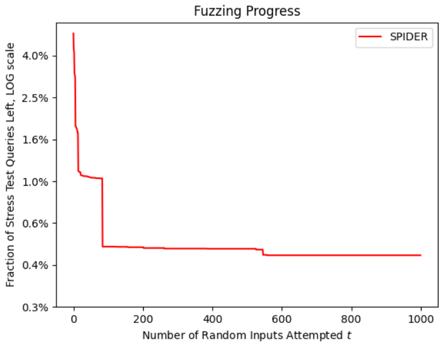


Figure 5: The progress of fuzzing (Section 4.2). The x-axis represents the number of random databases attempted (t), and the y-axis (re-scaled by log) is the fraction of neighbor queries left. y-value at $x = 0$ is the fraction of neighbors left after checking denotations on the database provided by Yu et al. (2018).

cases they happen to be semantically equivalent to the gold. We acknowledge that our fuzzing based approach has trouble distinguishing semantically close queries that differ only at a floating-point precision (e.g. "≤ `2.31`" vs. "< `2.31`"). Fortunately, however, we cannot find a false positive caused by this weakness in our subsequent manual evaluation.

Manual Evaluation. Even though our test suite achieves comprehensive code coverage, we still need to make sure that our method does not create any false positive on model-predicted queries. We focus on the queries from the 21 submissions that are considered incorrect by ESM but correct by our test suite evaluation, randomly sampled and manually examined 100 of them. *All* of them are semantically equivalent to the gold query; in other words, we did not observe a single error made by our evaluation method. We release these 100 model-predicted queries along with annotated reasons for why they are equivalent to the gold labels,[7] such that the research community can conveniently scrutinize the quality of our evaluation method.

We also confirm that our method can reliably evaluate model-predicted queries on WikiSQL (Zhong et al., 2017). We refer the readers to Appendix A.3 for further experimental details.

Difficulty	Mean	Std	Max
easy (%)	0.5 / 2.2	0.5 / 1.3	2.0 / 7.2
medium (%)	0.2 / 1.9	0.3 / 1.9	0.7 / 8.0
hard (%)	0.5 / 4.4	1.2 / 3.8	4.0 / 12.1
extra (%)	1.7 / 3.2	1.8 / 1.6	5.3 / 8.2
all data (%)	0.5 / 2.6	1.0 / 1.7	2.0 / 8.1

Table 1: The false positive/negative rate of the adapted exact set match metric (Section 5.3) for each difficulty split. We report the mean / std / max of these two statistics among 21 dev set submissions.

6.2 Errors of Traditional Metrics

Given that test suite evaluation empirically provides an improved approximation of semantic equivalence, we use test suite accuracy as ground truth and retrospectively examine how well ESM approximates semantic accuracy. We calculate the

[7] Manual examinations are available here.

Difficulty	Mean	Std	Max
easy (%)	3.6	1.2	6.0
medium (%)	5.9	0.9	8.2
hard (%)	8.0	1.5	10.3
extra (%)	11.0	3.5	17.6
all data (%)	6.5	1.0	9.0

Table 2: The false positive rate of single denotation accuracy (i.e., checking denotation only on the databases originally released in Yu et al. (2018)) for each difficulty split. We report the mean / std / max of these two statistics among 21 dev set submissions.

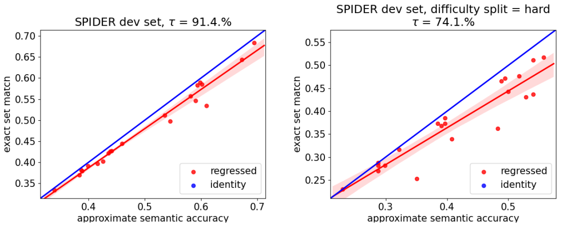
false positive/false negative rate for each difficulty split and report the mean, standard deviation, and max for all 21 submissions.

Table 1 shows the results. ESM leads to a nontrivial false negative rate of 2.6% on average, and 8.1% in the worst case. The error becomes larger for harder fractions of queries characterized by more complex queries. On the hard fraction, false negative rate increases to 4% on average and 12.1% in the worst case.

In Table 2, we report the difference between test suite accuracy and single denotation accuracy, which effectively means checking denotations of the model-predicted queries only on the databases from the original dataset release (Yu et al., 2018). In the worst case, single denotation accuracy creates a false positive rate of 8% on the entire development set, and 4% more on the extrahard fraction.

6.3 Correlation with Existing Metrics

Could surface-form based metric like ESM reliably track improvements in semantic accuracy?

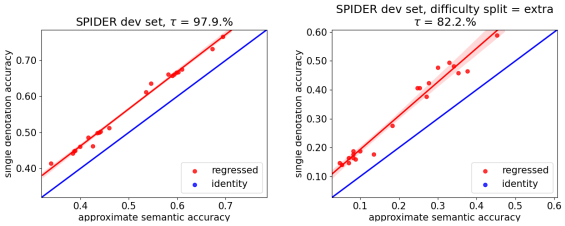


(a) τ = 91.4% on all queries in the dev set.

(b) τ = 74.1% on hard fraction of the dev set.

Figure 6: Kendall τ correlation between exact set match and test suite accuracy. Each dot is a dev set submission to the SPIDER leaderboard.

We plot ESM against test suite accuracy for all 21 dev set submissions in Figure 6. On a macro level, ESM correlates well with test suite accuracy with Kendall τ correlation 91.4% in aggregate; however,



(a) τ = 97.9% on all queries in the dev set.

(b) τ = 82.2% on extrahard fraction of the dev set.

Figure 7: Kendall τ correlation between single execution accuracy as originally defined in Yu et al. (2018) and test suite accuracy. Each dot is a dev set submission to the SPIDER leaderboard.

the correlation decreases to 74.1% on the hard fraction. Additionally, ESM and test suite accuracy starts to diverge as model performance increases. These two facts jointly imply that as models are becoming better at harder queries, ESM is no longer sufficient to approximate semantic accuracy. On a micro level, when two models have close performances, improvements in semantic accuracy might not be reflected by increases in ESM. On the hard fraction, 5 out of 21 submissions have more than four others that have lower test suite accuracy but higher ESM scores (i.e., there are five dots in Figure 6b such that for each of them, four other dots is located in its upper left).

Figure 7 plots the correlation between single denotation accuracy against test suite accuracy. On the extrahard fraction, four submissions have more than three others that have higher single denotation accuracy but lower test suite accuracy. Checking denotation only on the original database is insufficient.

We list the Kendall τ correlations between test suite accuracy and different metrics in Table 3 and plot them in the appendix Section A.4. The correlation with the current official metric is low without fixing the issue (3) identified in Section 5.3.

6.4 Computational Efficiency

On average, we distill 42 databases for each of the 1034 queries. In total, there are 695 databases since queries with the same database schema share the same test suite. These databases take 3.27GB in space (databases from the original datasets take 100.7MB). Running the gold queries on the entire test suite takes 75.3 minutes on a single CPU (compared to 1.2 minutes on the databases from the original datasets). Although test suite evaluation consumes more space and computational resources

Difficulty	Adapted	Official	Single Denot.
easy (%)	91	86	90
medium (%)	90	37	96
hard (%)	75	28	94
extra (%)	91	20	82
all data (%)	91	40	98

Table 3: Kendall τ correlation between various metrics and test suite accuracy across 21 submissions. **Adapted** refers to ESM after we fix the issue (3) identified in Section 5.3. **Official** refers to directly running the official evaluation script to evaluate, and **Single Denot.** refers to only checking denotation on the one database provided by Yu et al. (2018).

than single denotation evaluation, it is parallelizable and affordable by most researchers.

We may speed up the evaluation by checking denotation only on a single database sampled from the distribution $\mathcal{I}_g$. While this sped-up version sacrifices precision for speed, *retrospectively*, it produces the exact same outcomes as running the full test suite on the 21 submissions. Therefore, the sped-up version might be useful when occasional errors are tolerable (e.g. denotation based training). However, we still recommend using the full test suite for reliable evaluation, since a single sample from $\mathcal{I}_g$ cannot distinguish all neighbors, and checking denotations on multiple databases with comprehensive code coverage is always more reliable, especially when we have no prior information about the model-predicted queries.

7 Metrics Comparison and Analysis

We explain how ESM and test suite accuracy differ and provide representative examples (Figure 8).

False Positives Although standard ESM is strict, the adapted ESM (Section 5.3) can introduce false positives because it ignores table variable names. See Figure 8 row 1 for an example.

False Negatives Row 2-4 shows that slightly complicated queries usually have semantically equivalent variants, and it is nontrivial to tell whether they are semantically equivalent unless we execute them on a test suite or manually verify.

Nevertheless, even though test suite accuracy reliably approximates semantic accuracy according to our observation, researchers might also care about other aspects of a model-predicted query. Semantic accuracy is only concerned with *what* are the denotations of a query, but not *how* it calculates them. For example, Figure 8 row 5 represents one of the most common types of false negatives, where the model-predicted query chooses to join other tables even though it is unnecessary. While semantically correct, the model-predicted query increases running time. Figure 8 row 7 exhibits a similar but more complicated and rare example.

Inserting gold values into model-predicted queries as described in Section 5 might also unexpectedly loosen the semantic accuracy metric. For example, in Figure 8 row 6, the model-predicted query uses the "`LIKE`" keyword rather than the "=" operator. By SQL style conventions, "`LIKE`" usually precedes a value of the form "`%[name]%`" and corresponds to natural language query "`contains [name]`" rather than "`matches [name]`"; it seems plausible that the model does not understand the natural language query. However, if we replace the wrong value "`%[name]%`" with the gold value "`[name]`" after the "`LIKE`" operator, the predicate becomes semantically equivalent to "`= [value]`" and hence makes the query semantically correct. Value prediction is a crucial part of evaluating Text-to-SQL models.

8 Discussion and Conclusion

Semantic Evaluation via Test Suites We propose test suite accuracy to approximate the semantic accuracy of a Text-to-SQL model, by automatically distilling a small test suite with comprehensive code coverage from a large number of random inputs. We assure test suite quality by requiring it to distinguish neighbor queries and manually examining its judgments on model-predicted queries. Our test suites will be released for eleven datasets so that future works can conveniently evaluate test suite accuracy. This metric better reflects semantic accuracy, and we hope that it can inspire novel model designs and training objectives.

Our framework for creating test suites is general and only has two requirements: (1) the input is strongly typed so that the fuzzing distribution $\mathcal{I}_g$ can be defined and the sample input can be meaningfully executed, and (2) there exist neighbor queries N_g that are semantically close but different from the gold g. Since these two conditions hold in many execution environments, our framework might potentially be applied to other logical forms, such as λ-DCS (Liang, 2013), knowledge graphs (Lin et al., 2018), and python code snippets (Yin et al., 2018; Oda et al., 2015) if variable types can

	Error	Gold & Model Prediction	Explanation
1	False Positive	**Gold:** `SELECT T3.NAME, T2.COURSE FROM Course_arrange AS T1 JOIN Course AS T2 ON` `T1.COURSE_ID = T2.COURSE_ID JOIN Teacher AS T3 ON T1.TEACHER_ID = T3.TEACHER_ID;` **Prediction:** `SELECT T1.NAME, T2.COURSE FROM Course_arrange AS T1 JOIN Course AS T2 ON` `T1.COURSE_ID = T2.COURSE_ID JOIN Teacher AS T3 ON T1.TEACHER_ID = T3.TEACHER_ID;`	Exact set match does not account for table variable names.

2	False Negative	**Gold:** `SELECT TEMPLATE_ID FROM Templates`
		`EXCEPT SELECT TEMPLATE_ID  FROM Documents;`
		Prediction: `SELECT TEMPLATE_ID  FROM Templates`
		`WHERE TEMPLATE_ID NOT IN (SELECT TEMPLATE_ID FROM Documents);`
			"EXCEPT" is semantically equivalent to "NOT IN"

3	False Negative	**Gold:** `SELECT COUNT(*) FROM Area_code_state;` **Prediction:** `SELECT COUNT(STATE) FROM Area_code_state;`	Counting any column is the same.
4	False Negative	**Gold:** `SELECT TRANSCRIPT_DATE FROM Transcripts ORDER BY TRANSCRIPT_DATE DESC LIMIT 1;` **Prediction:** `SELECT MAX(TRANSCRIPT_DATE) FROM Transcripts;`	First element of descendingly sorted column is equivalent to maxing.
5	False Negative	**Gold:** `SELECT COUNT(*) FROM Cars_data WHERE HORSEPOWER > 150;` **Prediction:** `SELECT COUNT(*) FROM Cars_data as T1` `JOIN Car_names as T2 on T1.ID = T2.MAKEID where T1.HORSEPOWER > 150;`	Semantically correct redundant join.
6	False Negative	**Gold:** `SELECT AIRLINE FROM Airlines WHERE ABBREVIATION = "UAL";` **Prediction:** `SELECT AIRLINE FROM Airlines WHERE ABBREVIATION LIKE "UAL";`	If the string value is the same, "=" is equivalent to "LIKE"

7	False Negative	**Gold:** `SELECT LANGUAGE FROM Country_language`
		`GROUP BY LANGUAGE ORDER BY Count(*) DESC LIMIT 1;`
		Prediction: `SELECT Country_language.LANGUAGE FROM Country JOIN Country_language`
		`GROUP BY Country_language.LANGUAGE ORDER BY Count(*) Desc LIMIT 1;`
			The redundant join is implicitly a cross join, which will repeat every row in `Country_language` by [size of `Country` table] times. It leads to the same ranking if counted.

Figure 8: Representative examples where the exact set match (ESM) metric is different from test suite accuracy. False Positives happen when ESM judges a model-predicted query to be correct but test suite accuracy judges it to be wrong. False Negatives happen when the reverse takes place.

be heuristically extracted. We hope to see more future work that evaluates approximate semantic accuracy on the existing benchmarks and formulates new tasks amenable to test suite evaluation.

We do not attempt to solve SQL equivalence testing in general. While our test suite achieves comprehensive code coverage of the gold query, it might not cover all the branches of model-predicted queries. Adversarially, we can always construct a query that differs from the gold only under extreme cases and fools our metric. Fortunately, we never observe models making such pathological mistakes. However, it is necessary to revisit and verify this hypothesis some time later due to Goodhardt's law, since researchers will optimize over our metric.

Beyond Semantic Evaluation Although test suite evaluation provably never creates false negatives in a strict programming language sense, it might still consider "acceptable answers" to be wrong and result in false negatives in a broader sense. For example, in a database of basketball game results, the predicate "`A_wins`" is equivalent to "`scoreA > scoreB`" according to common sense. However, such a relation is not explicitly reflected in the database schema, and our procedure might generate an "unnatural" database where "`scoreA > scoreB`" but not "`A_wins`", hence distinguishing the model-predicted query from the gold. Fortunately, this issue is mitigated by current models. If "`A_wins`" is mentioned in the text, the model would prefer predicting "`A_wins`" rather than "`scoreA > scoreB`". Nevertheless, to completely solve this issue, we recommend future dataset builders to explicitly define the database generation procedure. Automatic constraint induction from database content and schema descriptions might also be possible, which is on its own an open research problem.

Additionally, some answers might be pragmatically acceptable but semantically wrong. For example, if a user asks "who is the oldest person?", the correct answer is a person's name. However, it also makes sense to return both the name and age columns, with the age column sorted in descending order. Collecting multiple gold SQL query references for evaluation (like machine translation) might be a potential solution.

Finally, as discussed in Section 7, there might be other crucial aspects of a model-predicted query beyond semantic correctness. Depending on the goal of the evaluation, other metrics such as memory/time efficiency and readability are also desirable and complementary to test suite accuracy.

Acknowledgement

We thank Wanyong Feng, Naihao Deng and Songhe Wang for rewriting queries and cleaning databases from Finegan-Dollak et al. (2018b). We thank Rishabh Agarwal, Tong Guo, Wonseok Hwang, Qin Lyu, Bryan McCann, Chen Liang, Sewon Min, Tianze Shi, Bailin Wang and Victor Zhong for responding to our cold email looking for WikiSQL model-predicted queries.

References

American fuzzy lop. http://lcamtuf.coredump.cx/afl. Accessed: 2020-5-12.

Automatic testing of haskell programs. https://hackage.haskell.org/package/QuickCheck-2.14/docs/Test-QuickCheck.html. Accessed: 2020-5-12.

Rishabh Agarwal, Chen Liang, Dale Schuurmans, and Mohammad Norouzi. 2019. Learning to generalize from sparse and underspecified rewards. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 130–140. PMLR.

P Ammann and J Offutt. Introduction to software testing. cambridge university press, 2008.

Ben Bogin, Jonathan Berant, and Matt Gardner. 2019a. Representing schema structure with graph neural networks for text-to-sql parsing. In *ACL*.

Ben Bogin, Matt Gardner, and Jonathan Berant. 2019b. Representing schema structure with graph neural networks for text-to-sql parsing. *arXiv preprint arXiv:1905.06241*.

DongHyun Choi, Myeong Cheol Shin, EungGyun Kim, and Dong Ryeol Shin. 2020. Ryansql: Recursively applying sketch-based slot fillings for complex text-to-sql in cross-domain databases. *https://arxiv.org/abs/2004.03125*.

Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic foundations and algorithms for deciding semantic equivalences of sql queries. *Proceedings of the VLDB Endowment*, 11(11):1482–1495.

Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An automated prover for sql.

Deborah A. Dahl. 1989. Book reviews: Computer interpretation of natural language descriptions. *Computational Linguistics*, 15(1).

Deborah A. Dahl, Madeleine Bates, Michael Brown, William Fisher, Kate Hunicke-Smith, David Pallett, Christine Pao, Alexander Rudnicky, and Elizabeth Shriberg. 1994. Expanding the scope of the atis task: The atis-3 corpus. In *HUMAN LANGUAGE TECHNOLOGY: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*.

Catherine Finegan-Dollak, Jonathan K Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. 2018a. Improving text-to-sql evaluation methodology. *arXiv preprint arXiv:1806.09029*.

Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. 2018b. Improving text-to-SQL evaluation methodology. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 351–360, Melbourne, Australia. Association for Computational Linguistics.

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

Hongyu Guo. 2017. A deep network with visual text composition behavior. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 372–377, Vancouver, Canada. Association for Computational Linguistics.

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-sql in cross-domain database with intermediate representation. *arXiv preprint arXiv:1905.08205*.

Pengcheng He, Yi Mao, K. Chakrabarti, and W. Chen. 2019. X-sql: reinforce schema representation with context. *ArXiv*, abs/1908.08113.

Wonseok Hwang, Jinyeung Yim, Seunghyun Park, and Minjoon Seo. 2019. A comprehensive exploration on wikisql with table-aware word contextualization. *ArXiv*, abs/1902.01069.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 963–973, Vancouver, Canada. Association for Computational Linguistics.

Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 254–265.

Fei Li and HV Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84.

Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc V Le, and Ni Lao. 2018. Memory augmented policy optimization for program synthesis and semantic parsing. In *Advances in Neural Information Processing Systems*, pages 9994–10006.

Percy Liang. 2013. Lambda dependency-based compositional semantics. *arXiv preprint arXiv:1309.4408*.

Xi Victoria Lin, Richard Socher, and Caiming Xiong. 2018. Multi-hop knowledge graph reasoning with reward shaping. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3243–3253, Brussels, Belgium. Association for Computational Linguistics.

Qin Lyu, Kaushik Chakrabarti, Shobhit Hathi, Souvik Kundu, Jianwen Zhang, and Zheng Chen. 2020. Hybrid ranking network for text-to-sql. *arXiv preprint arXiv:2008.04759*.

B. McCann, N. Keskar, Caiming Xiong, and R. Socher. 2018. The natural language decathlon: Multitask learning as question answering. *ArXiv*, abs/1806.08730.

Joan C Miller and Clifford J Maloney. 1963. Systematic mistake analysis of digital computer programs. *Communications of the ACM*, 6(2):58–63.

Sewon Min, Danqi Chen, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2019. A discrete hard EM approach for weakly supervised question answering. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2851–2864, Hong Kong, China. Association for Computational Linguistics.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584. IEEE.

Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 329–340.

Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*, IUI '03, page 149–157, New York, NY, USA. Association for Computing Machinery.

Tianze Shi, Kedar Tatwawadi, K. Chakrabarti, Yi Mao, Oleksandr Polozov, and W. Chen. 2018. Incsql: Training incremental text-to-sql parsers with nondeterministic oracles. *ArXiv*, abs/1809.05054.

Alane Suhr, Srinivasan Iyer, and Yoav Artzi. 2018. Learning to map context-dependent sentences to executable formal queries. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2238–2249. Association for Computational Linguistics.

Alane Laughlin Suhr, Kenton Lee, Ming-Wei Chang, and Pete Shaw. 2020. Exploring unexplored generalization challenges for cross-database semantic parsing. In *ACL 2020*.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers.

Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F. Wong, and Lidia S. Chao. 2019. Learning deep transformer models for machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1810–1822, Florence, Italy. Association for Computational Linguistics.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. Sqlizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–26.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pages 476–486. ACM.

Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, Vincent Zhang, Caiming Xiong, Richard Socher, Walter Lasecki, and Dragomir Radev. 2019a. CoSQL: A conversational text-to-SQL challenge towards cross-domain natural language interfaces to databases. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1962–1979, Hong Kong, China. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921.

Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, Emily Ji, Shreya Dixit, David Proctor, Sungrok Shim, Jonathan Kraft, Vincent Zhang, Caiming Xiong, Richard Socher, and Dragomir Radev. 2019b. SParC: Cross-domain semantic parsing in context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4511–4523, Florence, Italy. Association for Computational Linguistics.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2*, AAAI'96, page 1050–1055. AAAI Press.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.

A Appendix

A.1 Algorithmic Description of Section 4.2

Algorithm 1: Distilling a test suite S_g. N_g is the set of neighbor queries of g; $\mathcal{I}_g$ is a distribution of database inputs.

```
$S_g := \emptyset, N := N_g$ ;
for $t = 1, 2, \ldots 1000$ do
    $w_t \sim \mathcal{I}_g$;
    for $q \in N_g$ do
        if $D_{\{w_t\}}(q, g) = 1$ then
            $S_g.add(w_t)$;
            $N.remove(q)$
return $S_g$
```

A.2 Test Suite for Other Datasets

Data We download Academic, Advising, ATIS, Geography, IMDB, Restaurants, Scholar and Yelp from Finegan-Dollak et al. (2018a). For each dataset, we distill a test suite for the test split if it is already defined; otherwise we distill for the entire dataset.

We distill one shared test suite for the development set of Spider (Yu et al., 2018), CoSQL (Yu et al., 2019a) and SParC (Yu et al., 2019b), since they share the same 20 database schemata.

Test Suite Statistics The detailed test suite statistics can be seen in Table 4. The following list describes what each column represents:

- # Queries: the number of SQL queries we generate test suites for. Notice that this is not the full dataset size.
- Time: the time needed (in minute) to execute all the gold queries on its corresponding test suite on a single CPU. The smaller the value, the better.
- Size: the total size (in Giga-Bytes(G)/Mega-Bytes(M)) of the test suite. The smaller the value, the better.
- OrigSize: the size of the databases (in Giga-Bytes) in the original release.
- $|N_g|$: the average number of neighbor queries generated for each gold query g in the dataset.
- Undistinguished: the fraction of neighbor queries that cannot be distinguished by the test suite. The smaller the value, the better.
- # "Reliable": the estimated fraction of gold queries in a dataset that can be *reliably evaluated* (defined below). The larger the value, the better.

Spider, CoSQL and SParC have approximately the same statistics, since they share the same database schema and annotation convention. The other eight datasets have significantly longer queries with much more "`JOIN`" and "`WHERE`" operations. Hence, there are more spans to drop and more neighbors are generated per query.

Reliability Table 4 column "Undistinguished" implies that fuzzing cannot distinguish a non-trivial fraction of neighbor queries for some datasets. Besides cases where the neighbor queries are accidentally semantically equivalent to the gold, there are two major categories where fuzzing fails to distinguish semantically inequivalent neighbors.

- The gold query contains too many "`WHERE`" operations. For example, among the 93 queries in the ATIS test split, the maximum number of "`WHERE`" operations is 24 for a single query, whereas this number is only 2 among 1034 queries in the Spider development set. Distinguishing two queries that differ by only one "`WHERE`" operation is hard because the randomly sampled database needs to have a row that exactly satisfies all the "`WHERE`" clauses.
- The gold query contains predicates like "`WHERE COUNT(*) > 5000`". Distinguishing "`WHERE COUNT(*) > 5000`" from "`WHERE COUNT(*) > 4999`" requires the number of the target (intermediate) table to have a size exactly 5000. Such a requirement is particularly hard for randomly generated databases.

We say that a datapoint can be *reliably evaluated* if all of its undistinguished neighbors do not fall into the above two categories; then we estimate the fraction of datapoints that can be reliably evaluated for each dataset in Table 4. Fortunately, the majority of queries can be reliably evaluated for every dataset. Future manual efforts to hand-craft test suite might be needed to distinguish the neighbor queries and make test suite evaluation more reliable on ATIS and Advising.

Finally, Suhr et al. (2020) evaluates execution accuracy only on datapoints where the gold denotation is not empty. In comparison, at least one

Dataset	# Queries	Time ↓	Size ↓	OrigSize	$\|N_g\|$	Undistinguished ↓	# “Reliable” ↑
Spider	1034	75.3m	3.27G	0.10G	94	0.44%	100.0%
CoSQL	1007	75.6m	3.27G	0.10G	93	0.48%	100.0%
SParC	1203	86.7m	3.27G	0.10G	81	0.71%	100.0%
Academic	189	1.6m	0.03G	4.81G	368	1.36%	94.7%
Advising	76	1.7m	0.14G	0.03G	520	0.91%	63.2%
ATIS	93	19.2m	0.92G	0.06G	974	0.63%	76.3%
Geography	51	0.4m	2.21M	0.26M	108	5.28 %	88.2%
IMDB	97	0.8m	0.02G	0.99G	253	0.23%	100.0%
Restaurants	23	0.2m	1.37M	1.03M	379	0.14%	100.0%
Scholar	101	0.9m	9.43M	6.45G	107	0.54%	92.1%
Yelp	122	1.4m	0.02G	2.15G	274	0.07%	98.3%

Table 4: Detailed test suite statistics by datasets. Appendix Section A.2 includes detailed explanation of each column name. ↓/↑ means that we hope the number to be small/large. Spider, CoSQL and SParC share the same test suite.

database from our test suite produces non-empty gold denotation for every datapoint in all eleven datasets.

A.3 Evaluation on WikiSQL

We show that our test suite evaluation strategy also works well for model-predicted queries on WikiSQL (Zhong et al., 2017). The dev/test set contains 8420/15878 SQL queries, respectively.

Model-Predicted Queries We reached out to authors of individual works to obtain real model predictions on WikiSQL, and heard back from Min et al. (2019); McCann et al. (2018); Lyu et al. (2020); Hwang et al. (2019); He et al. (2019); Shi et al. (2018); Guo (2017); Agarwal et al. (2019); Liang et al. (2018); Wang et al. (2019).

We use the model-predicted queries from the first six works cited above since they provided model-predicted queries in the format consistent with Zhong et al. (2017), which can be easily converted into SQL queries. Specifically, we consider the model-predicted queries from the following eight models: MQAN unordered (McCann et al., 2018), X-SQL (He et al., 2019), HydraNet with/without Execution Guidance (Lyu et al., 2020), IncSQL (Shi et al., 2018), SQLova with/without Execution Guidance (Hwang et al., 2019) and HardEM (Min et al., 2019). This provides us in total (8420 + 15878) × 8 ≈ 200K model-predicted queries.

Test Suite Generation We run the fuzzing algorithm (Section 4) as before to create test suite. Since the most complex query in WikiSQL is simple and only consists of a single “WHERE” clause with an aggregation operation, our test suite can distinguish all the neighbors.

Metric Difference To check whether our test suite reliably evaluates semantic accuracy, we examine model-predicted queries where test suite accuracy disagrees with Exact Set Match (ESM) (as in Section 6.1).

We find that there is only one pattern where a semantically correct prediction is considered wrong by ESM: counting any column of a table gives exactly the same denotation. For example, “SELECT count(col1) from Table” is semantically equivalent to “SELECT count(col2) from Table” but different in surface form. After implementing a rule to filter out this equivalence pattern, we only find one model-predicted query that is considered wrong by ESM but correct by test suite accuracy, and we present it below.

The gold query is

```
SELECT MAX(col2) table WHERE col4 =
10;
```

, while the model-predicted query is

```
SELECT MAX(col2) FROM table WHERE
col2 > 10 AND col4 = 10;
```

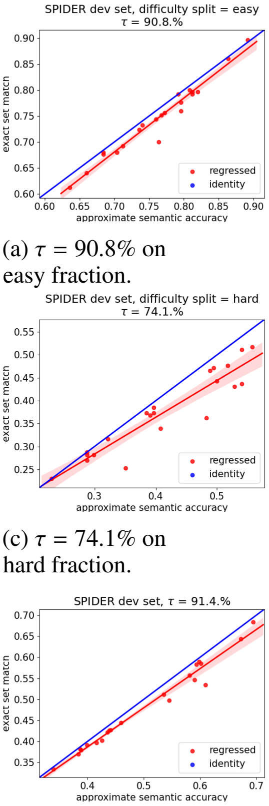
.This model-predicted query is not semantically equivalent to the gold, and hence our test suite evaluation makes an error. It over-generates a clause “WHERE col2 > 10” that is not covered by the test suite. None of our sampled database leads to a gold denotation fewer or equal to 10, which is a necessary and sufficient condition to distinguish these two queries.

To conclude, on WikiSQL, we only find 1 out of 200K model-predicted queries where our test suite accuracy makes an error, while we are able to verify that our test suite accuracy is correct for all the other model-predicted queries. This further implies that our method to develop semantic evaluation is robust and not dataset-specific.

On the other hand, however, the only semantically equivalent variant of the gold query in WikiSQL is replacing the column to be counted. Since we might still want to check which column the model-predicted query is counting for code readability, we do NOT recommend researchers to use test suite accuracy for WikiSQL.

A.4 Correlation Plot with Other Metrics

We plot the correlation between test suite accuracy and (1) adapted exact set match (Figure 9), (2) official SPIDER exact set match (Figure 10), and (3) single denotation accuracy (Figure 11) on each fraction of the difficulty split.



(a) τ = 90.8% on easy fraction.

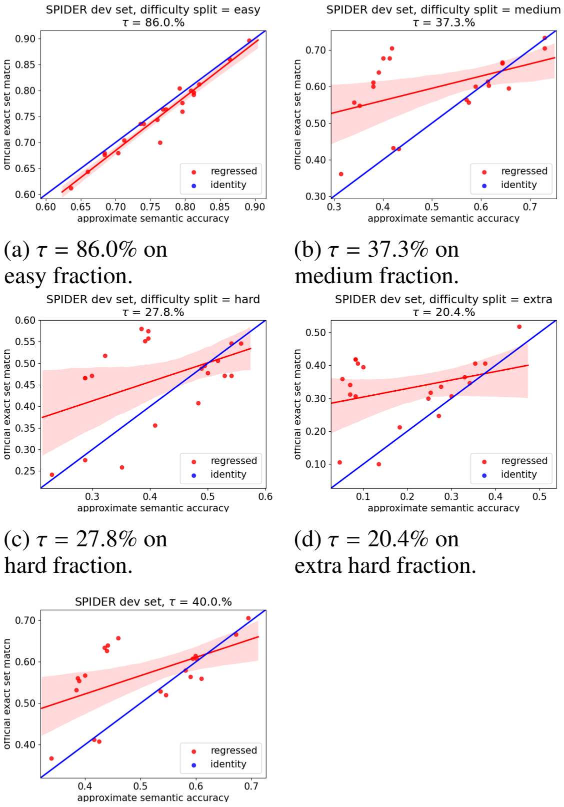
(b) τ = 90.1% on medium fraction.

(c) τ = 74.1% on hard fraction.

(d) τ = 91.0% on extra hard fraction.

(e) τ = 91.4% on all data.

Figure 9: Kendall τ correlation between **adapted exact set match** and fuzzing-based accuracy. Each dot in the plot represents a dev set submission to the SPIDER leader board.



(a) τ = 86.0% on easy fraction.

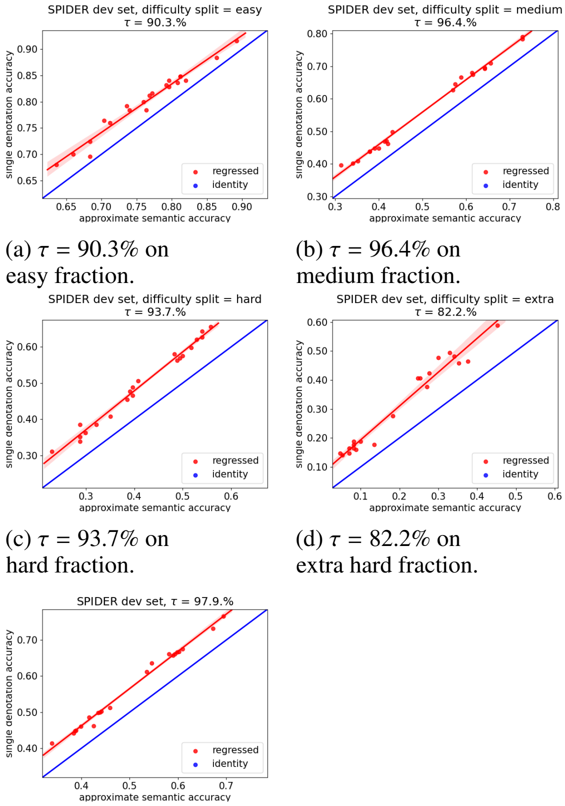
(b) τ = 37.3% on medium fraction.

(c) τ = 27.8% on hard fraction.

(d) τ = 20.4% on extra hard fraction.

(e) τ = 40.0% on all data.

Figure 10: Kendall τ correlation between the official **SPIDER exact set match** and fuzzing-based accuracy. Each dot in the plot represents a dev set submission to the SPIDER leader board.



(a) τ = 90.3% on easy fraction.

(b) τ = 96.4% on medium fraction.

(c) τ = 93.7% on hard fraction.

(d) τ = 82.2% on extra hard fraction.

(e) τ = 97.9% on all data.

Figure 11: Kendall τ correlation between **single denotation accuracy** and fuzzing-based accuracy. Each dot in the plot represents a dev set submission to the SPIDER leader board.