ParSEval: Plan-aware Test Database Generation for SQL Equivalence Evaluation

Chunyu Chen
Simon Fraser University
chunyu_chen@sfu.ca

Zhengjie Miao*
Simon Fraser University
zhengjie@sfu.ca

Yong Zhang
Huawei Technologies
yong.zhang3@huawei.com

Jiannan Wang
Huawei Technologies
Simon Fraser University
jnwang@sfu.ca

ABSTRACT

Deciding query equivalence has played an essential role in many real-world applications, including evaluating the accuracy of text-to-SQL models, where one needs to compare model-generated queries against ground truth queries. Although query equivalence is undecidable in general, researchers have developed two significant approaches to check query equivalence: formal verification-based and test-case-based. Verification-based solutions ensure correctness but may lack support for advanced SQL features and cross-database adaptability. Test cases are versatile but suffer from ad-hoc constraints and potential incorrectness (false positives).

In this paper, we propose ParSEval, a Plan-aware SQL Equivalence evaluation framework to generate test database instances for given queries. We observed that existing test data generation methods fail to fully explore the query structure. To address this limitation, ParSEval formally models specific behaviors of each query operator and considers all possible execution paths of the logical query plan by adapting the notion of branch coverage. We validated the effectiveness and efficiency of ParSEval on four datasets with AI-generated and human-crafted queries. The experimental results show that ParSEval supports up to 40% more query pairs than state-of-the-art verification-based approaches. Compared to existing test-case-based approaches, ParSEval reveals more non-equivalent pairs while being 21× faster.

PVLDB Reference Format:
Chunyu Chen, Zhengjie Miao, Yong Zhang, and Jiannan Wang. ParSEval: Plan-aware Test Database Generation for SQL Equivalence Evaluation. PVLDB, 18(11): 4750 - 4762, 2025.
doi:10.14778/3749646.3749727

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sfu-db/ParSEval.

1 INTRODUCTION

Determining the semantic equivalence of two SQL queries has long been an important task in the database community, with significant implications for query optimization [10, 36], query rewriting [43], and SQL assignment grading [12, 31]. Recently, this task has gained

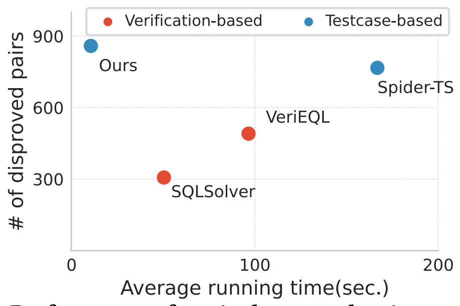


Figure 1: Performance of equivalence evaluation methods on 1534 pairs of queries from a text-to-SQL model for the BIRD benchmark [25]. # of disproved pairs is the number of pairs each method finds inequivalent.

increasing attention with the rise of text-to-SQL translation [5, 9, 32, 38, 40], where it plays a key role in evaluating the accuracy of text-to-SQL models. This scenario typically involves a set of natural language questions tied to a database schema and their corresponding ground truth SQL queries. The evaluation process then checks whether model-generated SQL queries are semantically equivalent to the ground truth queries.

The standard approach for evaluating the accuracy of text-to-SQL models involves executing the predicted queries on test databases — such as those in the Spider [38] and BIRD [25] benchmarks — and comparing their results with the ground truth query results. However, the evaluation process remains challenging, as query equivalence is undecidable in general [1], and there have been few efforts to improve the evaluation process in recent years. As we will show, while test-case-based methods have been popular and have driven advancements in text-to-SQL models, they still exhibit shortcomings by considering only a limited set of "neighboring queries," which can lead to *false positives* — failing to identify when a predicted query is not equivalent to the ground truth query.

Example 1. *Consider the following relation schema and natural language questions simplified from the BIRD benchmark [25]:*

Player(pid, pname, age) *stores information about football players and* PlayerAttributes(pid, rating) *stores information about their overall ratings. The user asks the text-to-SQL model to write a query to find players with the highest rating. The ground truth query Q_1, as shown below, uses a sub-query to determine if there are no higher ratings than the current player.*

```
Q1: SELECT p1.pid, p1.pname FROM Player AS p1
INNER JOIN PlayerAttributes AS p2 ON p1.pid = p2.pid
WHERE NOT EXISTS (
  SELECT * FROM PlayerAttributes p3
  WHERE p2.rating < p3.rating)
```

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749727

pid	pname	age	
001	Abby	24	r_1
125	Burnie	21	r_2
853	Chen	30	r_3

(a) `Player` relation

pid	rating	
001	8.2	s_1
125	8.2	s_2
853	6.0	s_3

(b) `PlayerAttributes` relation

Abby
Burnie

(c) Result of ground truth query Q_1

Abby

(d) Result of wrong query Q_2

Figure 2: An example instance D_1 to show that Q_1 and Q_2 in Example 1 return different results.

While the text-to-SQL model may predict a wrong query Q_2, which returns the first player after sorting the `PlayerAttributes` *table according to rating, and then joins the result with the* `Player` *table.*

```
Q2: SELECT p1.pid, pname FROM Player AS p1,
   (SELECT pid FROM PlayerAttributes
    ORDER BY rating DESC LIMIT 1) AS p2
WHERE p1.pid = p2.pid
```

As captured by the instance D_1 in Figure 2, Q_2 returns different results from Q_1 when there are ties for the top-rated player, as the `LIMIT 1` *operator selects only the first one and excludes others with the same highest rating. However, on the instance D_2 shown in Figure 3, generated by the previous approach [39] (where we replaced the random values with more readable ones), Q_1 and Q_2 return identical results.*

Previous test-case-based approaches focus on generating test cases to differentiate neighboring queries that vary by only a single predicate or operator. For instance, Spider-TS [39] employs a fuzzy testing method that randomly generates instances until all considered neighboring queries produce results different from the ground truth. However, this method often requires a longer run time compared to others, as shown in Figure 1. These approaches struggle to generate test cases that can distinguish queries whose differences do not arise from specific predicate values (like Q_1 and Q_2 in our example).

Formal verification offers another way to prove SQL query equivalence. We have seen several automatic equivalence-checking tools based on formal methods in recent years [13, 15, 20, 35, 42]. Some of these tools [12, 20] generate small symbolic inputs, execute queries symbolically, and iteratively enlarge inputs to detect differences. However, these methods face two key limitations: (i) limited SQL feature support, and (ii) inefficiency on certain queries due to the vast query input space (see Figure 1 and the example below).

Example 2. *Consider the following query Q_3 generated by a text-to-SQL model to answer the question in Example 1. It uses the aggregate function* `MAX` *to find the highest overall rating among all players and select players whose ratings are equal to this highest value.*

```
Q3: SELECT p1.pid, p1.pname FROM Player AS p1
INNER JOIN PlayerAttributes AS p2 ON p1.pid = p2.pid
WHERE p2.rating = (
    SELECT MAX(rating) FROM PlayerAttributes)
```

Q_3 is equivalent to the ground truth query Q_1; however, prior verification-based methods [15, 20, 35] fail to prove this equivalence within a timeout threshold (360 seconds in our experiment).

These issues prevent the adoption of the verification-based approaches in evaluating text-to-SQL models (see results in Section 5), and people have been using the test-case-based approaches for automatic text-to-SQL evaluation [39]. However, as shown in Example 1, existing methods may lead to many false positives.

pid	pname	age	
001	Abby	19	r_4
760	Burnie	26	r_5
566	Chen	25	r_6
515	Dolores	24	r_7

(a) `Player` relation

pid	rating	
001	6.4	s_4
515	8.1	s_5
566	8.6	s_6

(b) `PlayerAttributes` relation

Figure 3: An instance D_2 that fails to distinguish Q_1 and Q_2.

Therefore, we aim to improve the test-case-based approach by incorporating formal verification techniques to better capture the query semantics. Our solution is also inspired by software testing techniques that create test cases based on *path coverage* [4], whose goal is to cover all execution paths in a program's control-flow graph. In the context of testing SQL queries, a query Q's execution paths depend on its physical plan, which may bear little resemblance to the query syntax, and the same query syntax may lead to very different execution plans (e.g., sort-merge join vs. hash join). To address this, we define the execution paths of an SQL query Q based on its logical plan and consider different behaviors of each query operator as the branches during the execution of Q.

Our contributions. Our contributions are summarized below.

- We propose a framework for describing plan coverage in test database instances for SQL queries by adapting path coverage to query plans, with various coverage constraints capturing different query operator behaviors.
- Using plan coverage, we define the test suite generation problem with different coverage requirements. We argue that the plan coverage can be an indicator of the test case's capability to differentiate queries.
- We encode operator coverage constraints as symbolic expressions and generate test instances by solving them. To address limited feature support and inefficiencies with large expressions, we introduce a hybrid algorithm that speculatively assigns concrete values to improve efficiency.
- We implement our test instance generation algorithms in ParSEval and evaluate their efficiency and effectiveness using LLM-generated and human-crafted queries. ParSEval detects at least 8% more inequivalent queries than existing test-case-based methods while running 21× faster. Compared to verification-based methods, it supports up to 40% more query pairs and achieves a 7× speedup.

2 PRELIMINARIES

In this paper, we consider a wide range of SQL queries that contain common operators such as selection, projection, join, set/bag operations, and group-by aggregation. We also consider more advanced SQL features, including `ORDER BY`, `CASE WHEN`, `IN/NOT IN`, etc. More formally, for a database instance D and a query Q, we use $Q(D)$ to denote the output of Q on D. Let Γ denote a set of integrity constraints on the schema of D; we write $D \models \Gamma$ to denote that D comply with Γ. We consider the following standard integrity constraints: keys, foreign keys, unique, and not null.

2.1 Query Equivalence and Text-to-SQL Evaluation

Let $\mathbf{R}$ be a database schema that contains m relations $R_1, R_2, \cdots, R_m$. Let $\mathbf{A}$ be the set of all attributes in $\mathbf{R}$. Each relation R_i is defined on a subset of attributes $A_i \subseteq \mathbf{A}$. Let $\mathrm{dom}(A)$ be the domain of attribute $A \in \mathbf{A}$. Given the database schema $\mathbf{R}$, let D be a given database of $\mathbf{R}$, and let the corresponding relations of $R_1, R_2, \cdots, R_m$ be $R_1^D, R_2^D, \cdots, R_m^D$, where R_i^D is a collection of tuples defined on $dom(A_i)$. The input size of database D is denoted as $|D| = \sum_{1 \le i \le m} |R_i^D|$. Let Q_1 and Q_2 be two queries, $\mathbf{R}$ be a database schema, and Γ be a set of integrity constraints. We say that Q_1 and Q_2 are *semantically equivalent*, denoted $Q_1 \equiv_{\mathbf{R},\Gamma} Q_2$, if, over every database instance of $\mathbf{R}$ that complies with Γ, they return identical results. When there exists a database instance D' of the schema $\mathbf{R}$ and $D' \models \Gamma$ such that $Q_1(D') \neq Q_2(D')$, we are sure that $Q_1 \not\equiv_{\mathbf{R},\Gamma} Q_2$, and we call D' a counterexample for Q_1 and Q_2.

Definition 1 (Test Suite Equivalence). *A test suite S is a set of database instances $\{D_1, D_2, \cdots, D_{|S|}\}$ of a given schema $\mathbf{R}$ that comply with a set of integrity constraints Γ. For two queries Q_1 and Q_2, we say that Q_1 and Q_2 are S-equivalent, denoted as $Q_1 \equiv_S Q_2$, if $Q_1(D_i) = Q_2(D_i)$ on every $D_i \in S, 1 \le i \le |S|$. The test suite equivalence result of Q_1 and Q_2 is a (1) True Positive: if $Q_1 \equiv_S Q_2 \wedge Q_1 \equiv_{\mathbf{R},\Gamma} Q_2$, (2) False Positive: if $Q_1 \equiv_S Q_2 \wedge Q_1 \not\equiv_{\mathbf{R},\Gamma} Q_2$, (3) True Negative: if $Q_1 \not\equiv_S Q_2 \wedge Q_1 \not\equiv_{\mathbf{R},\Gamma} Q_2$.*

We say that a test suite S distinguishes two queries Q_1 and Q_2 if there exists a counterexample $D_i \in S$ such that $Q_1(D_i) \neq Q_2(D_i)$. In the context of text-to-SQL evaluation, a benchmark involves a set of n pairs $\{(Q_{nl}^i, Q_g^i) \mid 1 \le i \le n\}$, where Q_{nl}^i is a question in natural language, and Q_g^i is the corresponding ground truth query. A text-to-SQL model $\mathtt{M}$ is evaluated on the benchmark by writing a query $\mathtt{M}(Q_{nl}^i, \mathbf{R})$ ($\mathtt{M}(Q_{nl}^i)$ for short) for each Q_{nl}^i.

Example 3. *Continue with the schema, natural language question, queries, and database instances in previous examples. The pair of question and ground truth query, $(Q_{nl}^1, Q_g^1) =$ ("find the player with the highest rating", Q_1) is an example text-to-SQL benchmark. Suppose that a text-to-SQL model $\mathtt{M}_1$ returns $\mathtt{M}_1(Q_{nl}^1) = Q_2$, another model $\mathtt{M}_2$ returns $\mathtt{M}_2(Q_{nl}^1) = Q_3$, a test suite $S_1 = \{D_1\}$, and a test suite $S_2 = \{D_1, D_2\}$. Since Q_1 and Q_3 are equivalent, we have $Q_1 \equiv_{S_1} Q_3$ and $Q_1 \equiv_{S_2} Q_3$; however, we have $Q_1 \equiv_{S_1} Q_2$ but $Q_1 \not\equiv_{S_2} Q_2$.*

By definition, semantic equivalence implies test suite equivalence, i.e., for two queries Q_1, Q_2, and a test suite S with the schema $\mathbf{R}$ and integrity constraints Γ, we have $Q_1 \equiv_{\mathbf{R},\Gamma} Q_2 \implies Q_1 \equiv_S Q_2$, while the reverse does not hold. Hence, test suite equivalence approximates semantic equivalence, and such approximation will introduce only *False Positives* (two queries are S-equivalent for a given test suite S but not semantically equivalent) but no *False Negatives* (it is impossible that two queries are not S-equivalent but semantically equivalent). In contrast, $Q_1 \equiv_S Q_2$ may still hold for two inequivalent queries (False Positive is possible).

Definition 2 (Test Suite Generation Problem). *Given a pair of queries Q_1 and Q_2, schema $\mathbf{R}$, and integrity constraints Γ, the goal of the Test Suite Generation Problem is to find a test suite S of the schema $\mathbf{R}$ such that $D \models \Gamma$ for all $D \in S$ and the difference between test suite equivalence and semantic equivalence, $\mathbb{1}[Q_1 \equiv_S Q_2] - \mathbb{1}[Q_1 \equiv_{\mathbf{R},\Gamma} Q_2]$, is minimized.*

In the context of text-to-SQL evaluation, we would like to prepare a test suite such that for each $\{(Q_{nl}, Q_g)\}$ pair, it can find as many queries that are not semantically equivalent to Q_g as possible:

$$\begin{array}{ll} \textit{maximize} & \mathbb{1}(Q_g \not\equiv_{\mathbf{R},\Gamma} \mathtt{M}(Q_{nl})) \cdot \mathbb{1}(Q_g \not\equiv_S \mathtt{M}(Q_{nl})) \\ \textit{s.t.} & S = \{D_1, \ldots, D_{|S|}\} \\ \textit{where} & D_j \models \Gamma, j = 1, \ldots, |S| \end{array} \tag{1}$$

Note that the objective above only considers *True Negative* because when query pairs are given, the term of *True Positives* ($\mathbb{1}[Q_g \equiv_{\mathbf{R},\Gamma} \mathtt{M}(Q_{nl})] \cdot \mathbb{1}[Q_g \equiv_S \mathtt{M}(Q_{nl})]$) will be constant (since $\mathbb{1}[Q_g \equiv_{\mathbf{R},\Gamma} \mathtt{M}(Q_{nl})] = \mathbb{1}[Q_g \equiv_S \mathtt{M}(Q_{nl})] = 1$).

However, it is infeasible to enumerate all possible queries written by models in practice. Therefore, prior test-case-based approaches consider neighbor queries of the ground truth query by mutations in query predicates and operators [31, 33, 39] to enumerate the candidate query space. Next, we will introduce these practical strategies.

2.2 Coverage-Guided Testing

In software testing, test coverage measures the percentage of software code that is executed during the tests [4]. For SQL, the DBMS evaluates a SQL query differently from how it is rewritten syntactically (the physical query plan vs. the query syntax), making coverage-guided testing for SQL more challenging than imperative or functional programming languages. There are also cases where one operator's evaluation affects another in the query (e.g., the actual sorting algorithm depends on the output size of the operator below sorting). Hence, the variety of physical plans poses unnecessary complexities for testing query equivalence, and prior works [31, 39] focus on testing selection predicates or single operators. For example, state-of-the-art test suite generation method used in text-to-SQL benchmarks [39] considers *Clause Coverage*, i.e., for each clause $\mathbf{C}$ in the selection or join condition in the form of $x\ op\ y$ or $x\ op\ c$ where x, y are attributes in relations, c is a constant, and op is a binary operator, a test database instance D is said to cover the clause $\mathbf{C}$ if, when evaluating on D, queries with mutant clauses of $\mathbf{C}$ return different results than the original query.

Example 4. *Consider the query below on the same* `Player` *and* `PlayerAttributes` *tables as in Example 1 that finds all players with an age below 25 and a rating above 8.*

```
Q4: SELECT p1.name FROM Player p1, PlayerAttributes p2
WHERE p1.age < 25 AND p2.rating > 8 AND p1.pid = p2.pid
```

Clause Coverage considers mutants of each clause in the query, e.g., changing the condition `p1.age < 25` *to* `p1.age ≥ 25`, `p1.age < 24` *or other mutants with neighboring constant numbers and comparison operators; or changing* `p2.rating > 8` *to clauses like* `p2.rating ≥ 8` *and* `p2.rating > 8.1`. *The test database instance should contain players that satisfy different clause mutants and hence differentiate queries with these mutants from the original Q_4.*

Similar to *Clause Coverage*, prior methods also consider *Single Operator Coverage* [31]. For an operator `O` in a ground truth query Q, a test instance D is said to cover `O` if Q and the mutant query Q' that replaces `O` with a mutant operator `O'` return different results on D. These methods consider each clause or operator independently and, hence, fail to exercise complex predicates with disjunctions. They either lack a formal discussion on the space of query mutants/neighbors or encode query mutants in an ad-hoc way.

2.3 Modeling Query Semantics

Next, we introduce how to model the semantics of the query. Unlike existing test data generation methods that consider the *syntax* of queries or the *semantics* of only one operator, equivalence provers [13, 15, 20, 36, 42] model the semantics of entire queries to prove query equivalence. These provers transform a SQL query into a symbolic expression, which encodes the query output using K-relations [18] or the follow-up U-semirings [11], to model a relation and capture the *multiplicity* of tuples in a relation.

In this paper, we base our modeling of query semantics on a recent extension of U-semirings by Ding et al. [15], which leads to a state-of-the-art SQL equivalence solver that composes the U-semiring expression (U-expression for short) from each relational query operator. Under U-semiring, a relation is modeled as a function that maps tuples to a commutative semiring $(\mathcal{U}, \mathbf{0}, \mathbf{1}, +, \times)$. Extending K-relations with $\|\cdot\|, \texttt{not}(\cdot)$ and the unbounded summation, U-semiring is able to model the deduplication operator, negation, and projection, and thus models aggregate functions and set/bag operations under bag semantics. Table 1 lists some example U-expressions for relational operators. For a predicate ϕ, the U-expression $[\phi]$ is defined by $[\phi] = 1$ (if ϕ holds) or $[\phi] = 0$ (otherwise). For a tuple t and a set of attributes A, $t.A$ denotes the values of attributes A of t. For more details of the semiring frameworks, readers can refer to prior works [11, 18].

Example 5. *Consider again query Q_4 from Example 4, which finds players under age 25 with overall ratings above 8. Its logical plan is shown in Figure 4. The U-expressions for the two selection operators are $f_1(t) := [\![p_1]\!](t) \times [t.age < 25]$ and $f_2(t) := [\![p_2]\!](t) \times [t.rating > 8]$, resp., where $[\![p_1]\!](t)$ and $[\![p_2]\!](t)$ return the tuple's multiplicity in relations p_1 and p_2. During evaluation, we ground the symbols with concrete values. For example, on the instance D_1 in Figure 2, $f_1(t)$ evaluates to 1 for the only two tuples r_1 and r_2 that have age values less than 25, since their multiplicities (i.e., $[\![p_1]\!](r_1)$ and $[\![p_1]\!](r_2)$) in the base relation* `Player` *are 1. Similarly, $f_2(t)$ evaluates to 1 for two tuples s_1 and s_2 with ratings above 8.*

Then the U-expression for the inner join is denoted as $f_3(t) := \sum_{t_1,t_2}[t = t_1 \cdot t_2] \times f_1(t_1) \times f_2(t_2) \times [t1.pid = t2.pid]$. On the instance D_1, the join operator returns two tuples, $r_1 \cdot s_1$ and $r_2 \cdot s_2$, and $f_3(r_1 \cdot s_1) = f_1(r_1) \times f_2(s_1) \times [r_1.pid = s_1.pid] = 1$, $f_3(r_2 \cdot s_2) = f_1(r_2) \times f_2(s_2) \times [r_2.pid = s_2.pid] = 1$. Finally the projection is encoded as $f_4(t) := \sum_{t_1} f_3(t_1) \times [t = t1.name]$.

In this way, the execution of logical query operators is symbolically encoded as U-expressions. Hence, we can define the "code coverage" of SQL queries based on the logical plan and the corresponding U-expressions.

3 MODEL FOR QUERY COVERAGE

In this section, we present the desiderata of our query coverage model in the ParSEval framework. The model formalizes *plan coverage* based on operator semantics, and we generate test instances by solving constraints derived from different plan coverage criteria.

3.1 Semantic Operator Coverage

As mentioned in section 2.2, existing test suite generation methods only consider *Clause Coverage* for selection predicates or Single Operator Coverage defined in an ad-hoc way, which leads to false positives in query equivalence evaluation. To address this, we draw inspiration from prior work [8, 17, 27] and propose to look into how the queries are evaluated from bottom to top in the logical query plan and consider all possible ways in which each query operator is evaluated. Unlike CInsGen [17, 27], which only considers *possible ways in which one query returns non-empty results*, our approach also accounts for cases when query operators fail to return results, which can lead to more test instances to help distinguish the mutant queries from the ground truth query. Similarly, QAGen [8] focuses on generating instances that satisfy the multiplicity constraints in the query (e.g., `COUNT(*) > 1000`) or specified by the user. On the contrary, our approach considers not only satisfying the multiplicity constraints (which is essential for the query to return non-empty results) but also the cases when these constraints are not satisfied.

To model how a query operator evaluates, inspired by the encoding of query semantics using U-expressions (as shown in Table 1) in existing equivalence provers [11, 15], we propose the notion of *semantic coverage* of a query operator `O`, which considers:

(1) *Predicate:* if `O` has a predicate ϕ, e.g., when evaluating a selection operator on a relation R, the evaluation of `O` may trigger the "positive branch" when ϕ evaluates to true and the "negative branch" when ϕ evaluates to false. For predicates with disjunctions, multiple ways can trigger each branch.
(2) *Multiplicity:* if `O` considers the multiplicity of tuples (deduplication, set/bag operation, group-by, aggregation), changing the multiplicity of an input tuple may make a difference.
(3) *Grouping:* if `O` involves grouping tuples (projection, group-by), i.e., when evaluating `O`, the output tuple is derived from a group of input tuples, and the number of groups and the size of each group may make a difference.
(4) *Binary:* if `O` has two input relations R_1 and R_2, the evaluation of `O` depends on how the multiplicity of an output tuple is derived: (i) from both R_1 and R_2 (inner join, intersection, union, difference), (ii) from only one input (difference, union), or (iii) from the cases that require the non-existence of the (part of the) output tuple in one input (set difference, outer join).

Considering all these above aspects, ParSEval divides the evaluation of each operator into a few cases and we can define *Coverage Constraints* and *Semantic Operator Coverage* in a systematic way.

Definition 3 (Coverage Constraints). *Let Q be a query plan tree rooted at an operator `O`, R be the input relation to `O` (or R_1, R_2 when `O` is binary), R_{out} be the output relation of `O`, a coverage constraint β for `O` is a U-expression in the format of*

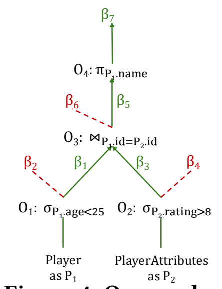
(1) (Predicate) $\sum_{t \in R}[\phi'(t)] \times R(t) > 0$ or $\sum_{t_l \in R_1, t_r \in R_2}[\phi'(t_l, t_r)] \times R_1(t_l) \times R_2(t_r) > 0$ where ϕ' is a variant of the selection or join condition ϕ.
(2) (Multiplicity) $\sum_{t \in R_{out}}[R_{out}(t) > c] > 0$ where $c \in \mathcal{N}$ is a parameter.
(3) (Grouping) $\sum_{t \in R_{out}} \|R_{out}(t)\| > c$ where $c \in \mathcal{N}$ is a parameter.
(4) (Binary-nojoin) $\sum_{t_l \in R_1}[\sum_{t_r \in R_2}[\phi(t_l, t_r)] \times R_1(t_l) \times R_2(t_r) = 0] > 0$, where ϕ is the join condition; may swap R_1 and R_2.
(5) (Binary-set/bag) $\sum_{t_l \in R_1} \sum_{t_r \in R_2}[t_l = t_r] \times [R_1(t_l)\ \mathsf{op}_1\ 0] \times [R_2(t_r)\ \mathsf{op}_2\ 0] \times [R_1(t_l)\ \mathsf{op}_3\ R_2(t_r)] > 0$ where $\mathsf{op}_1, \mathsf{op}_2, \mathsf{op}_3$ are comparison operators; may skip the last clause $[R_1(t_l)\ \mathsf{op}_3\ R_2(t_r)]$.

Table 1: Example of U-expression definitions [11, 15, 36] for part of the query operators considered in this paper.

Query Operator	U-expression notation	Explanation	Example SQL query	Example U-expression $f(t)$
Relation R	$[\![R]\!](t)$	$[\![R]\!](t)$ is a function that returns the multiplicity of tuple t in relation R.	`SELECT * FROM R`	$[\![R]\!](t)$
Selection $\sigma_\phi(R)$	$[\![R]\!](t) \times [\phi(t)]$	$\phi(t)$ denotes encoding the predicate ϕ with tuple t; $[b]$ returns 1 if the predicate b is true. Otherwise, $[b]$ returns 0.	`SELECT * FROM R WHERE R.x < 500`	$[\![R]\!](t) \times [t.x < 500]$
Projection $\pi_L(R)$	$\sum_t f(t)$	$\sum_t f(t)$ returns the sum of $f(t)$ for possible tuple $t \in R$, where f is a function: $tuple \to \mathbb{N}$.	`SELECT R.x FROM R`	$\sum_{t_1}([t = t_1.x] \times [\![R]\!](t_1))$
Inner Join $R \bowtie_\phi S$	$\sum_{t_1,t_2} f_l(t_1) \times f_r(t_2) \times \phi(t_1, t_2)$	$e_1 \times e_2$ denotes the product of two U-expressions.	`SELECT * FROM R JOIN S ON R.x0 = S.y0`	$\sum_{t_l,t_r}([t = t_l \cdot t_r] \times [\![R]\!](t_l) \times [\![S]\!](t_r) \times [t_l.x0 = t_r.y0] \times \texttt{not}([\texttt{IsNull}(tl.x0)]))$

Table 2: Coverage categories and example coverage constraints for each operator type.

Operator type and example	Categories	Example Coverage Constraints	Explanation
Selection: $\sigma_\phi(R)$	Predicate	positive: $\sum_t R(t) \times [\phi(t)] > 0$, negative: $\sum_t R(t) \times (\neg\phi(t)) > 0$	There exists a tuple passes condition ϕ; there exists a tuple does not.
Join ($\bowtie$, $⟖$, $⟕$, $⟗$, $\times$): $R \bowtie S$	Predicate, Binary	positive: $\sum t_l \sum_{t_r} [R(t_l) \times S(t_r) \times \phi(t_l, t_r)] > 0$, negative: $\sum t_l \sum_{t_r} R(t_l) \times S(t_r) \times [\neg\phi(t_l, t_r)] > 0$	There exists a pair of tuples from R and S that can join; there exists a tuple from R that does not join with any tuple in S.
Projection, Deduplicate (π_L, δ): $\delta(R)$	Multiplicity, Grouping	positive: $\sum_t [(\sum_{t'} [t = t'] \times R(t')) > 1] > 0$, $\sum_t \Vert \sum_{t'} [t = t'] \times R(t') \Vert > 1$	Before deduplication, there exists one tuple has duplicates; there exists at least two distinct tuples/
Set/Bag operators ($\cup, \cap, -, \uplus, ⩀, \ominus$): $R - S$	Multiplicity, Binary	positive: $\sum_t [R(t) > 0] \times [S(t) = 0] > 0$, negative: $\sum_t [R(t) > 0] \times [S(t) > 0] > 0$	There exists one tuple in R but not in S; there exists one tuple in both R and S.
Group-by Aggregation: ($\gamma_{\mathsf{L},\mathsf{agg(y)}}(R)$)	Multiplicity, Grouping	positive: $\sum_t [(\sum_{t'} [t.L = t'.L] \times R(t')) > \mathsf{gs}] > 0$	There exists one group whose size is greater than the threshold.



$$\beta_1 : \sum_t [\![P_1(t)]\!] \times [t.age < 25] > 0$$
$$\beta_2 : \sum_t [\![P_1(t)]\!] \times [t.age \geq 25] > 0$$
$$\beta_3 : \sum_t [\![P_2(t)]\!] \times [t.rating > 8] > 0$$
$$\beta_4 : \sum_t [\![P_2(t)]\!] \times [t.rating \leq 8] > 0$$

$$f_1(t_1) = [\![P_1(t_1)]\!] \times [t_1.age < 25]$$
$$f_2(t_2) = [\![P_2(t_2)]\!] \times [t_2.rating > 8]$$
$$f_3(t_1, t_2) = [t_1.pid = t_2.pid]$$
$$\beta_5 : \sum_{t_1,t_2} f_1(t_1) \times f_2(t_2) \times f_3(t_1, t_2) > 0$$
$$\beta_6 : \sum_{t_1} [\sum_{t_2} f_1(t_1) \times f_2(t_2) \times f_3(t_1, t_2) = 0] > 0$$
$$\beta_7 : \text{same as } \beta_5 \text{ (returns non-empty results)}$$

Figure 4: Query plan and example cov. constraints for Q_4.

Moreover, we say a coverage constraint β for operator $\mathtt{O}$ *represents a (i)* positive branch *if it represents a case where* $\mathtt{O}$ *returns a non-empty result or (ii)* negative branch *otherwise.*

Figure 2 shows the coverage constraints of some relational operators. Intuitively, a coverage constraint encodes how an operator evaluates over certain input tuple(s). Operators like projection, group-by, and aggregation always return non-empty results, so there are no negative branches in these cases. In contrast, binary operators are more complex: generating output tuples requires satisfying certain multiplicity and binary constraints. For example, consider bag-difference ($R \ominus S$), a tuple t would appear in the result as long as its multiplicity in R is greater than that in S; while for the negative branches, the tuple t will be excluded when its multiplicity in R is smaller than or equal to that in S, or it only exists in S.

Furthermore, given a concrete database instance D and a U-expression E derived from a query Q, we can evaluate E on D (denoted as E^D) in a bottom-up fashion, the same as when evaluating the logical query plan; for the given instance D, the domain for every summation $\sum_t$ is fixed (t iterates over either a base table or the result table of an operator in Q).

Example 6. *Consider again the query Q_4 in Example 4 and its logical plan shown in Figure 4. For each query operator, example coverage constraints in U-expressions are in the right side of Figure 4. For instance, to make the selection operator* $\mathtt{O}_1$ *return a non-empty result, an input instance should contain a tuple that passes the selection condition age < 25, captured by satisfying the constraint β_1 on the "positive branch"; the "negative branch" is denoted by β_2. To evaluate these constraints for* $\mathtt{O}_1$ *on D_1 from Figure 2, we iterate over all tuples in the* `Player` *table, and it turns out that we have $[\beta_1]^{D_1} = 1$ (r_1: [001, "Abby", 24] and r_2: [125, "Burnie", 21] pass the selection predicate) and $[\beta_2]^{D_1} = 1$ (r_3: [853, "Chen", 30] does not pass the selection predicate). Similarly, for $\mathcal{B}_{\mathtt{O}_2}$, we have $[\beta_3]^{D_1} = [\beta_4]^{D_1} = 1$. For the join operator, the constraint β_5 on the "positive branch" indicates that there are tuples from both sides that can join. Similarly, β_6 on the "negative branch" represents the case where one tuple exists in the left relation that cannot join with any tuple in the right relation. By iterating over all tuples output by the selection operators, we have $[\beta_5]^{D_1} = 1$ and $[\beta_6]^{D_1} = 0$, since both tuples r_1, r_2 from* $\mathtt{O}_1$ *can join with a tuple from* $\mathtt{O}_2$.

Definition 4 (Semantic Operator Coverage). *Let Q be a query plan tree rooted at an operator* $\mathtt{O}$, *D be a database instance, $\mathcal{B}_\mathtt{O}$ be a set of coverage constraints, the semantic operator coverage $\mathtt{CovOpr}_{\mathcal{B}_\mathtt{O}}(\mathtt{O}, D)$ for an operator* $\mathtt{O}$ *of D is the set of pairs $\{(\beta, l) \mid \beta \in \mathcal{B}_\mathtt{O}\}$ where $l \in \{0, 1\}$ indicates the evaluation result of $[\beta]^D$.*

Example 7. *Following Example 6 and let $\mathcal{B}_{\mathtt{O}_1} = \{\beta_1, \beta_2\}$, $\mathcal{B}_{\mathtt{O}_2} = \{\beta_3, \beta_4\}$, $\mathcal{B}_{\mathtt{O}_3} = \{\beta_5, \beta_6\}$ be the sets of constraints we consider for operators $\mathtt{O}_1, \mathtt{O}_2, \mathtt{O}_3$ in the query plan, resp. We have the semantic operator coverage $\mathtt{CovOpr}_{\mathcal{B}_{\mathtt{O}_1}}(\mathtt{O}_1, D_1) = \{(\beta_1, 1), (\beta_2, 1)\}$, $\mathtt{CovOpr}_{\mathcal{B}_{\mathtt{O}_2}}(\mathtt{O}_2, D_1) = \{(\beta_3, 1), (\beta_4, 1)\}$, $\mathtt{CovOpr}_{\mathcal{B}_{\mathtt{O}_3}}(\mathtt{O}_3, D_1) = \{(\beta_5, 1), (\beta_6, 0)\}$.*

Note that *Clause Coverage* and mutant operator coverage in prior works [31, 39] can be regarded as special cases of *Semantic Operator Coverage*. For example, Spider-TS [40] generates test instances by considering predicates ϕ' that differ in one clause from the selection predicate ϕ in the ground truth query.

With the logical plan of a query and the definition of semantic operator coverage, we can now define the plan coverage of a test database instance for a query. Although a query may have many different equivalent logical plans (e.g., one may push down a selection operator to the bottom of a plan), the choice of the specific logical plan does not affect our coverage model. In our implementation, ParSEval obtains the logical plan using Calcite [7].

Definition 5 (Query Plan Coverage). *Let Q be a query plan tree, D be a database instance, $\mathcal{B} = \bigcup_\mathtt{O} \mathcal{B}_\mathtt{O}$ be a set of coverage constraints on each operator in Q, the query plan coverage $\mathtt{Cov}_\mathcal{B}(Q, D)$*

for Q of D with regard to $\mathcal{B}$ is the union of the semantic operator coverage $\texttt{CovOpr}_{\mathcal{B}_0}(\texttt{O}, D)$ for each operator $\texttt{O}$ in Q.

When it is clear from the context, we omit the set of coverage constraints in the notation above and use $\texttt{Cov}(Q, D)$ and $\texttt{CovOpr}(\texttt{O}, D)$.

3.2 Coverage-based Test Suite Generation

With the plan coverage definition on a single database instance, we can now define the coverage of a test suite.

Definition 6 (Coverage of Test Suites). *Let Q be a query, S be a set of database instances $\{D_1, D_2, \ldots, D_{|S|}\}$, and $\mathcal{B}$ be the set of coverage constraints, the coverage for Q of S with respect to $\mathcal{B}$, is the set of $\{\texttt{Cov}(Q, D_1), \ldots, \texttt{Cov}(Q, D_{|S|})\}$ (each $\texttt{Cov}(Q, D_i)$ is a set of (β, l) pairs as in Definition 4, where β is a coverage constraint for an operator in Q and l is the evaluation result of β on the instance D_i).*

Example 8. *Continue with Example 7, let $\mathcal{B}$ be the set of all coverage constraints in Figure 4, the query plan coverage $\texttt{Cov}(Q_4, D_1)$ for Q_4 of D_1 in the running example is $\{(\beta_1, 1), (\beta_2, 1), (\beta_3, 1), (\beta_4, 1), (\beta_5, 1), (\beta_6, 0), (\beta_7, 1)\}$, and $\texttt{Cov}(Q_4, D_2) = \{(\beta_1, 1), (\beta_2, 1), (\beta_3, 1), (\beta_4, 1), (\beta_5, 1), (\beta_6, 1), (\beta_7, 1)\}$. Thus, a test-suite consisting of D_1 and D_2 will have coverage $\{\texttt{Cov}(Q_4, D_1), \texttt{Cov}(Q_4, D_2)\}$ (set of two sets).*

Recall that in Definition 2, the goal of the Test Suite Generation problem is to minimize the difference between the Test Suite accuracy and semantic accuracy, which is further reduced to the maximization of the number of *True Negatives* discovered by the test suite. In other words, since test suites provide the guarantee that when two queries return different results, they must not be equivalent, we would like to focus on disproving the equivalence of as many (ground truth query, predicted query) pairs as possible. Nevertheless, there is no direct way to solve the optimization problem unless we can enumerate all possible SQL queries and have a symbolic execution engine that supports all of them. To this end, to efficiently generate test suites that maximize the objective, we propose using the query plan coverage as an indicator of the test suite's capability to capture inequivalent queries. Now, the optimization is to maximize the total number of different coverages for each ground truth query Q_g.

$$\begin{array}{lll} \textit{maximize} & |\texttt{Cov}(S, Q_g)| & \\ \textit{s.t.} & S = \{D_1, \ldots, D_{|S|}\} & \quad (2) \\ \textit{where} & D_j \models \Gamma, j = 1, \ldots, |S| & \end{array}$$

In other words, ideally, we would like to find a complete test suite that satisfies as many as different subsets of coverage constraints for each ground truth query. However, the number of test instances and the size of each instance matter in terms of execution time. Hence, we would like to find small test suites.

Relaxed completeness. While the above optimization goal allows us to find a complete set of test suites by considering every possible combination of the coverage constraints of the ground truth query, the number of database instances will be exponential to the number of coverage constraints, leading to significant computational overhead. Therefore, to keep a small yet comprehensive set of test database instances, we may consider individual coverage constraints instead of the combination of coverage constraints to relax the requirement of the completeness of the coverage constraints. This gives us a different optimization objective:

pid	pname	age	mul.
x_1	p_1	a_1	m_1
x_2	p_2	a_2	m_2
$x_1 \neq x_2$			

(a) `Player` relation (P_1)

pid	rating	mul.
y_1	r_1	n_1
y_2	r_2	n_2
$y_1 \neq y_2$		

(b) `PlayerAttributes` relation (P_2)

Figure 5: An example symbolic instance I_1.

$$\begin{array}{lll} \textit{minimize} & |S| & \\ \textit{s.t.} & \bigcup_{j=1}^{|S|} \{\beta | (\beta, 1) \in \texttt{Cov}(Q_g, D_j)\} = \mathcal{B} & \quad (3) \\ \textit{where} & S_i = \{D_1, \ldots, D_{|S|}\}, D_j \models \Gamma, j = 1, \ldots, |S| & \end{array}$$

The new optimization problem in Equation (3) considers each coverage constraint *individually* and only requires that each coverage constraint is satisfied by one test instance in the test suite. We may obtain a solution to the optimization by post-processing the solution to Equation (2), which requires solving a set cover problem over the set of instances in the solution to Equation (2). However, such a post-processing solution does not improve the efficiency.

4 PLAN-AWARE TEST DATA GENERATION

In this section, we will present practical algorithms for constructing instances to satisfy coverage constraints using constraint solving.

4.1 Overview

For a query plan tree Q, Figure 2 gives examples of different evaluation branches for each operator to help consider possible behaviors of query operators. We aim to construct a test suite in which each instance satisfies different combinations of coverage constraints. Prior works like Spider-TS [39] only consider clause coverage and use fuzzy testing, leading to accuracy and performance issues. To address this, as illustrated in Figure 6, ParSEval parses the query to obtain U-Expressions, and then encodes the resulting coverage constraints into SMT constraints over variables in a symbolic instance. Then, we can populate the database instance with values from the solution to the SMT constraints.

Symbolic Instance [12, 22]. A symbolic table T_i with a relational schema R_i is a collection of tuples where each tuple $t \in T_i$ and each attribute $A \in Attr(R_i)$, $t[A]$ is either a symbolic variable, a constant from $Dom(A)$, or the `NULL` value. Each t is annotated with a *multiplicity* attribute, which is the formula that evaluates to a non-negative integer. Moreover, T_i is also annotated with symbolic constraints in the form of First Order Logic (FOL) expressions that tuples in T_i must satisfy, for example, the `unique` or `key` constraints. A symbolic instance consists of a collection of symbolic tables and symbolic constraints similar to those in each T_i, like foreign key constraints. Figure 5 shows an example of a symbolic instance, where the key constraints are encoded by $x_1 \neq x_2$ and $y_1 \neq y_2$.

Evaluating Queries on Symbolic Tables. While U-expressions can be regarded as symbolic representations that encode query outputs using input tuples (see Table 1), it is not straightforward to derive the outputs from base tables. Hence, to generate symbolic formulas from coverage constraints, we built a symbolic execution engine to obtain symbolic tables as query results. In certain cases (e.g., base table, selection, deduplication, etc.), the result table is a copy of the input table with the *multiplicity* formula updated the same way as in Table 1. However, in some cases, we may be

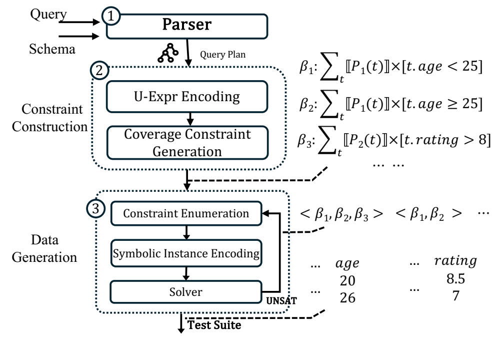
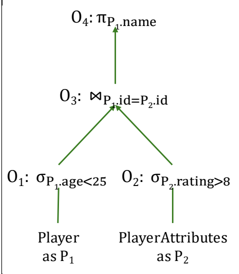

Query
Schema
1
Parser
Query Plan
2
U-Expr Encoding
Coverage Constraint Generation
Constraint Construction
$\beta_1: \sum_t [\![P_1(t)]\!] \times [t.age < 25]$
$\beta_2: \sum_t [\![P_1(t)]\!] \times [t.age \geq 25]$
$\beta_3: \sum_t [\![P_2(t)]\!] \times [t.rating > 8]$
… …
3
Constraint Enumeration
Symbolic Instance Encoding
Solver
UNSAT
Data Generation
$< \beta_1, \beta_2, \beta_3 >$ $< \beta_1, \beta_2 >$ …
… age … rating
20 8.5
… 26 … 7
Test Suite

Figure 6: Workflow of ParSEval.


O_4: $\pi_{P_1.name}$
O_3: $\bowtie_{P_1.id=P_2.id}$
O_1: $\sigma_{P_1.age<25}$
O_2: $\sigma_{P_2.rating>8}$
Player as P_1
PlayerAttributes as P_2

pname	mul.
p_1	$m_{41} : m_{31} + m_{32} + (m_{33} + m_{34}) \times [p_1 = p_2]$
p_2	$m_{42} : (m_{33} + m_{34}) \times [p_1 \neq p_2]$

(a) T_4**:Result of** O_4

pid	pname	age	pid	rating	mul.
x_1	p_1	a_1	y_1	r_1	$m_{31} : m_{11} \times m_{21} \times [x_1 = y_1]$
x_1	p_1	a_1	y_2	r_2	$m_{32} : m_{11} \times m_{22} \times [x_1 = y_2]$
x_2	p_2	a_2	y_1	r_1	$m_{33} : m_{12} \times m_{21} \times [x_2 = y_1]$
x_2	p_2	a_2	y_2	r_2	$m_{34} : m_{12} \times m_{22} \times [x_2 = y_2]$

(b) T_3**: Result of** O_3

pid	pname	age	mul.
x_1	p_1	a_1	$m_{11} : m_1 \times [a_1 < 25]$
x_2	p_2	a_2	$m_{12} : m_2 \times [a_2 < 25]$

(c) T_1**: Result of** O_1

pid	rating	mul.
y_1	r_1	$m_{21} : n_1 \times [r_1 > 8]$
y_2	r_2	$m_{22} : n_2 \times [r_2 > 8]$

(d) T_2**:Result of** O_2

Figure 7: Intermediate symbolic execution results of Q_4 **on** I_1**.**

unable to directly evaluate U-expressions when reasoning about query outputs with a fixed-size symbolic instance. For operators like projection and group-by that involve summation, it can be tricky as the way to denote "possible tuples" is to iterate over all tuples in the input relation, and each output tuple's *multiplicity* formula depends on the comparison results with other tuples. Moreover, for set/bag operators, the result tuple's *multiplicity* formula depends on both input relations, which requires enumerating all tuples in the second relation, while the U-expression itself is very simple: we can use the U-expression $[R(t) > 0] \times [S(t) = 0]$ to denote if a tuple t exists in the results of $R - S$, but in symbolic tables, one has to iterate over all tuples in S and compare them with the tuple t.

Example 9 (Symbolic execution of Q_4). *Continue with query Q_4 and its logical plan in Figure 4. When evaluating Q_4 on symbolic instance I_1, the output of each operator is also a symbolic table, as shown in Figure 7. Note that the results of selections (T_1 and T_2) directly come from adding the selection predicate to the multiplicity annotation, and the result of the join (T_3) comes from the cross product of T_1 and T_2, where each output's multiplicity combines the multiplicity of both input tuples and the join condition. The result of projection (T_4) first "merges" the first two tuples in T_3 ($m_{31} + m_{32}$) and the last two tuples in T_3 ($m_{33} + m_{34}$), since they share the same* pname *variable. Then, we further add the term $(m_{33} + m_{34}) \times [p_1 = p_2]$ to $m_{31} + m_{32}$ in the multiplicity of tuple (n_1). This step is identical to the unbounded summation in a U-expression, where we sum all tuples that share the same projected attribute values. However, the multiplicity of tuple (p_2) is different: when $p_1 = p_2$, we should not have (p_2) in the result.*

4.2 ParSEval-Sym: A Bottom-up Approach by Symbolic Execution

In this part, we introduce a basic approach ParSEval-Sym to modeling a SQL query with U-expression and generating a set of instances with different coverage. ParSEval-Sym (the procedure in Algorithm 1) takes a schema and a query as input, with a parameter *InsSizeLimit* setting the maximum number of tuples per table.

Algorithm 1 first constructs coverage constraints via BuildConstraints, which implements the query plan traversal strategies from Section 3.1 to encode constraints. We omit the details of BuildConstraints, since the constraints can be derived based on Definition 5 and Figure 2 during the traverse of the query plan tree.

After constructing the coverage constraints, Algorithm 1 enumerates all possible subsets of uncovered constraints in $\mathcal{B}_n$ starting from bigger subsets (Line 5). It then initializes the symbolic instance with increasing size. The next challenge is that no existing tools can evaluate the U-expression on instances. Hence, we introduce a symbolization procedure that translates U-expressions into logical formulas, explicitly encoding coverage constraints with symbols. Algorithm 2 demonstrates this procedure, which first obtains the symbolic execution results of Q and then translates the coverage constraints into symbolic constraints. We omitted the details of SymbolicExec and TranslateConstraint, which encode U-expressions into SMT expressions with input symbols. Note that ParSEval-Sym translates the U-expression predicate $[p]$ to an expression $\texttt{ITE}(p, 1, 0)$, returning 1 if p is True and 0 otherwise.

The resulting formulas can be solved by SMT solvers such as Z3 [14] and cvc5 [6]. Additionally, we encode database constraints for each table into symbolic constraints in the solver's input (line 8). If the solver returns SAT, it also provides a model for the constraints, yielding concrete values for each variable in the symbolic instance.

Example 10 (Running ParSEval-Sym on Q_4). *Consider the query Q_4 and the coverage constraints in Figure 4 again. First, the algorithm constructs the sets of positive and negative constraints, which are labeled green and red, resp. in Figure 4: $\mathcal{B}_p = \{\beta_1, \beta_3, \beta_5\}$, $\mathcal{B}_n = \{\beta_2, \beta_4, \beta_6\}$ (β_7 omitted since in this case it is the same as β_5).*

Then, the algorithm enumerates the uncovered subset of $\mathcal{B}_n$, starting from the maximum subset, $\mathcal{B}_n$ itself. Suppose InsSizeLimit = 2 for a simple illustration. To find if there exists a database instance that can satisfy all constraints in $\mathcal{B}_p$ and $\mathcal{B}_n$, the algorithm needs to encode the coverage constraints using a symbolic instance, whose size is set to 1 initially (i.e., each table in the symbolic instance has one tuple). However, it is not possible to satisfy any negative constraints with an instance when the size is 1, since the instance has to satisfy the positive constraints (e.g., there cannot be a tuple t in the Player *table that satisfies both $t.age < 25$ and $t.age \geq 25$).*

Then, the size of the symbolic instance increments to 2 (e.g., I_1 in Figure 5), and now through the EncodeConstraints *function in Algorithm 2, β_1, β_2 on the selection operator O_1 using I_1. We obtain symbolic constraints φ_1, φ_2:*

$\varphi_1 : m_1 \times \texttt{ITE}(a_1 < 25, 1, 0) + m_2 \times \texttt{ITE}(a_2 < 25, 1, 0) > 0,$

$\varphi_2 : m_1 \times \texttt{ITE}(a_1 \geq 25, 1, 0) + m_2 \times \texttt{ITE}(a_2 \geq 25, 1, 0) > 0$

Similarly, for β_3 and β_4 of the other selection operator O_2, we obtain symbolic constraints φ_3, φ_4. For the join operator O_3, we encode β_5 and β_6 using the input relations obtained from calling the SymbolicExec *function (which are the output of O_1 and O_2):*

$\varphi_5 = \sum_{1 \leq i \leq 2} \sum_{1 \leq j \leq 2} m_i \times n_j \times \texttt{ITE}(a_i < 25, 1, 0) \times \texttt{ITE}(r_j > 8, 1, 0) \times \texttt{ITE}(x_i = y_j, 1, 0) > 0$ *(at least one pair of tuples can join)*

$\varphi_6 = \sum_{1\le i\le 2} \mathtt{ITE}(\sum_{1\le j\le 2} m_i \times n_j \times \mathtt{ITE}(a_i < 25, 1, 0) \times \mathtt{ITE}(r_j > 8, 1, 0) \times \mathtt{ITE}(x_i = y_j, 1, 0) = 0, 1, 0) > 0$ *(at least one tuple in* `Player` *cannot join with any tuples in* `PlayerAttributes` *after the selection).*

The solver then takes the conjunction of $\varphi_1, ..., \varphi_6$ and the database constraints as input, and it fails to find a model — we cannot obtain an instance with size 2 that satisfies all the constraints. Since we have reached the size limit, the algorithm explores the next maximum unvisited subset of $\mathcal{B}_n$, $\{\beta_2, \beta_4\}$. This time, the solver finds a solution (see below) to the conjunction of $\varphi_1, ..., \varphi_5$ and the database constraints.

pid	*pname*	*age*
001	*Abby*	*20*
002	*Burnie*	*26*

(a) **`Player`** ***relation*** *(P_1)*

pid	*rating*
001	8.5
002	7.0

(b) **`PlayerAttributes`** ***relation*** *(P_2)*

In the next iteration, the algorithm explores the remaining uncovered subset β_6 and builds the corresponding symbolic constraints for the solver, which returns a solution as below.

pid	*pname*	*age*
001	*Abby*	*20*
002	*Burnie*	*24*

(a) **`Player`** ***relation*** *(P_1)*

pid	*rating*
001	8.5
002	7.0

(b) **`PlayerAttributes`** ***relation*** *(P_2)*

Note that in the first instance, β_6 is not satisfied because only the rows with pid = 001 pass the selection condition; while in the second instance, <002, Burnie, 24> passes the selection condition in `O`$_1$*, but no tuples output by* `O`$_2$ *can join with this tuple.*

As shown in the above example, the Algorithm 1 follows a greedy heuristic that prioritizes the largest subsets of negative constraints first, rather than exhaustive enumeration. In contrast, to find an exact solution to the relaxed optimization problem in Section 3.2, one would need to enumerate all subsets of coverage constraints, identify satisfiable ones, and then solve a minimum set cover problem over the valid instances. In practice, the greedy heuristic performs well, as most subsets of the coverage constraints are satisfiable. Our ablation study results (see Section 5.4) confirm the efficiency and accuracy of the simplified procedure.

4.3 ParSEval-Hybrid: Optimization by a hybrid approach

ParSEval-Sym simulates how a query executes with a U-expression and explores all feasible execution paths represented by coverage constraints. It enumerates tuples in the symbolic instance to encode constraints, but as query complexity or instance size increases, the symbolic expressions grow more complex. For example, to cover different branches in Figure 4, ParSEval-Sym increases the size of tuples in each table, which can lead to path explosion due to the lack of a principled method for handling unbound summations.

Moreover, although U-expression models SQL queries under bag semantics [15], it still struggles with complex query features like `ORDER BY` and `aggregate` functions. For instance, consider a common type of query, `<SELECT ... ORDER BY rating DESC OFFSET 2>` to find players with the second-highest rating. Suppose we have two tuples, t_1 and t_2, in the `PlayerAttributes` table. To handle the sort operator, we use `ITE` functions to represent comparisons in the multiplicity of each result tuple: $mul(t_1) \times \mathtt{ITE}(t_1.rating \ge t_2.rating, 0, 1)$ and $mul(t_2) \times \mathtt{ITE}(t_1.rating < t_2.rating, 0, 1)$. As the number of tuples grows, these expressions become increasingly complex, making it inefficient to solve the constraints. These limitations significantly affect the performance of ParSEval-Sym.

Algorithm 1 ParSEval-Sym

```
BuildInstance-Symbolic($\mathbf{R}, Q, InsSizeLimit$)
    Input: $R$: database schema; $Q$: query;
    Output: A list of database instances with different coverage for $Q$.
1   instances = [ ]
2   $\mathcal{B}_p, \mathcal{B}_n$ = BuildConstraints($Q$)
3   visited = $\emptyset$, covered = $\emptyset$
4   while covered $\neq \mathcal{B}_n$
5       $\Delta$ = the maximum unvisited subset of $\mathcal{B}_n$, where covered $\cap \Delta = \emptyset$
6       for $n$ = 1 $\rightarrow$ $InsSizeLimit$
7           $D$ = Initialize($\mathbf{R}$, n)
8           $\varphi$ = DBCons($D$) $\wedge \bigwedge_{\delta \in \Delta \cup \mathcal{B}_p}$ EncodeConstraints($\delta, D$)
9           if IsSAT($\varphi$)
10              covered = covered $\cup\Delta$, instances.append(BuildIns($\varphi$))
11              break
12      visited = visited $\cup\{\Delta\}$
13  return instances
```

4.3.1 Speculative value assignment. To this end, we propose a novel hybrid approach that speculatively assigns a concrete value to symbols in the translated symbolic expression before calling the solver. There are two main intuitions behind this approach. First, although it uses a complex symbolic expression to encode the semantics of query operators like projection and group-by, which consider all cases of input tuples, we may only need to include limited cases when generating the test database instance. Second, for certain predicates, we can evaluate their values quickly by directly assigning concrete values to the symbols in the predicate when there is high confidence that the assignment would be a valid solution.

For example, φ_5 in Example 10 involves the summation over four product-only symbolic expressions with multiple `ITE` functions, whose complexity could be reduced by pre-determining the evaluation results of certain clauses considering the key-foreign key relationship. By modeling the semantics of the join operator, the symbolic execution considers all possible combinations of input tuples, leading to the four sub-U-Expressions in φ_5; however, we are sure that at most two of the $\mathtt{ITE}(x_i = y_j, 1, 0)$ will evaluate 1 since $P_2.pid$ is a foreign key referring to $P_1.pid$. Therefore, we can speculatively determine $x_1 = y_1$ and then $x_2 = y_2$ and decide that $x_1 = y_2$ and $x_2 = y_1$ must be false, then directly removing the two sub-expressions involving $x_1 = y_2$ and $x_2 = y_1$. Hence, we simplify the constraint φ_5 to $m_1 \times n_1 \times \mathtt{ITE}(a_1 < 25, 1, 0) \times \mathtt{ITE}(r_1 > 8, 1, 0) + m_2 \times n_2 \times \mathtt{ITE}(a_2 < 25, 1, 0) \times \mathtt{ITE}(r_2 > 8, 1, 0) > 0$, largely reducing the size of the symbolic expression.

We implemented the speculative value assignment optimization to reduce the size of symbolic expressions and accelerate the solving of coverage constraints. This optimization slightly alters the procedure in Algorithm 2: when encountering operators with optimization opportunities, the constraint-building procedure replaces one symbol with another or generates concrete values randomly for the attributes involved in these operators. For example, for unique attributes like pid, assigning random values to symbols does not affect the evaluation of other constraints, as pid values are only used in join conditions and key constraints. Later, when solving the constraints using an SMT solver, these concrete values reduce the solution space. However, if the speculative assignment fails, we

Algorithm 2 Encode coverage constraints with a symbolic instance.

```
EncodeConstraints$(Q, I)$
    Input: $Q$: a query plan tree where the root node is annotated with
    coverage constraint $\beta$; $I$: a symbolic database instance.
    Output: $\varphi$: a symbolic constraint.
    if $\beta$ is on the output of $Q$
        $R_1$ = SymbolicExec$(Q, I)$
        $\varphi$ = TranslateConstraint$(\beta, R_1, None)$
    else
        $R_1$ = SymbolicExec$(Q.lchild, I)$
        if $Q.root \in \{\sigma, \pi, \delta, \gamma\}$ // unary operator
            $\varphi$ = TranslateConstraint$(\beta, R_1, None)$
        else // binary operator
            $R_2$ = SymbolicExec$(Q.rchild, I)$
            $\varphi$ = TranslateConstraint$(\beta, R_1, R_2)$
    return $\varphi$
```

keep the original symbolic expression and rerun the solver. Below, we detail the different strategies for assigning values in other cases.
Handling group-by aggregation. When encountering group-by aggregations, we can assign constants to the symbols of group-by attributes and pre-determine the group size. Without this assignment, the U-expression of aggregation would have to consider all cases where the tuples form a group, leading to an exponential increase in the size of the expression. For instance, there can be $O(2^n)$ cases for grouping n tuples (i.e., all n tuples in one group, splitting them into two groups, then three groups, and so on), but we don't have to build instances for all these $O(2^n)$ cases in practice.

Therefore, as long as there are no other constraints like selection predicates on the group-by attributes, this value assignment allows us to pre-define the grouping results and eliminate redundant comparisons between different tuples (which is expressed as an "unbounded summation" in the U-Expression).

5 EXPERIMENT

In this section, we empirically compared ParSEval against state-of-the-art verification-based and testcase-based approaches. We used query pairs (ground truth vs. LLM-generated/human-crafted queries) from established benchmarks. In particular, our experiments aim to answer the following research questions: 1) How well does ParSEval support different SQL features? 2) How effective is ParSEval in finding non-equivalent query pairs? 3) How does the running time of ParSEval compare to that of the state-of-the-art systems? 4) Can ParSEval produce better test suites than existing ones in text-to-SQL benchmarks?

5.1 Experimental Setup

We implemented ParSEval in Python 3.8, leveraging the query parser in Apache Calcite [7]. All experiments were conducted locally on a 64-bit Ubuntu 18.04 LTS server with a 32-core CPU (2.00GHz Intel(R) Xeon(R)) and 1000GB of 2666MHz DDR4 RAM. We compared ParSEval with state-of-the-art verification-based and test-case-based methods that are publicly available. Unless otherwise specified, ParSEval refers to ParSEval-Hybrid, the optimized version of our approach described in Section 4.3.

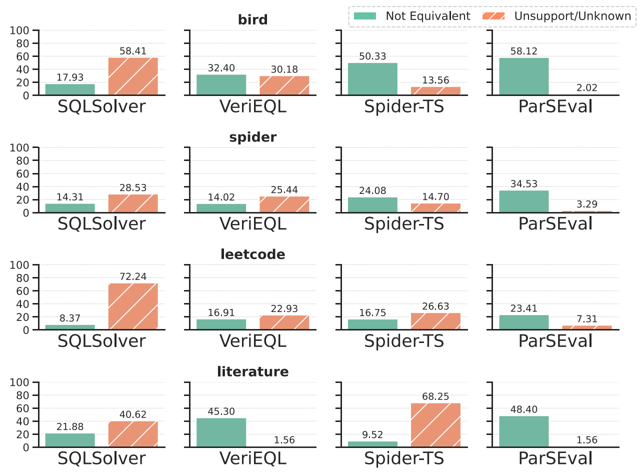


Figure 8: Ratio of disproved and unsupported/unknown pairs in different datasets.

- **SQLSolver [15]** is a state-of-the-art verification-based prover that translates SQL queries into LIA logic and evaluates the query equivalence via an SMT-Solver.
- **VeriEQL [20]** is a bounded equivalence checker that uses symbolic execution. It finds an instance to disprove query equivalence by incrementally increasing the size of each relation.
- **Spider-TS [39]** generates test database instances randomly and disproves two queries by comparing their execution results. It has been used in the Spider text-to-SQL benchmark [38].

There is another well-known test-case-based framework, XData [31]. However, we did not include it as a baseline due to its limited support ratio reported in [20], which might be affected by outdated maintenance. There are also other formal verification-based provers, such as SPES [42] and QED [35]; we did not include them in the following experiments because they lack the support for foreign key constraints in their open-sourced codebase, and most databases in the benchmarks we use involve foreign key constraints. If we ignore the foreign key constraints, these approaches would introduce *false negatives*.

All experiments used a 360-second timeout, as we observed no significant increase in the number of disproved query pairs beyond this threshold in our preliminary experiments. Note that the formal-verification-based approaches take two queries as input, while the test-case-based approaches take one query. For a fair comparison, we generated test data for both the ground truth query and the predicted query when running Spider-TS and ParSEval.
Datasets. We used query pairs from four publicly available benchmarks, two for text-to-SQL and two for human-crafted queries.

- **BIRD [25]:** BIRD is a popular benchmark for text-to-SQL targeting real-world application scenarios. We used the dev set containing 1534 ground truth queries across 11 databases, and the LLM-generated queries are from the repository of DAIL-SQL [9].
- **Spider [38]:** Spider is another text-to-SQL benchmark. We used the dev set with 1034 ground truth queries across 20 databases. The LLM-generated queries are from the same repo as above [9].
- **Leetcode:** This dataset is collected by He et al. [20] from answers to LeetCode questions. We used 23,865 of its query pairs after removing the `UPDATE` and `DELETE` statements.

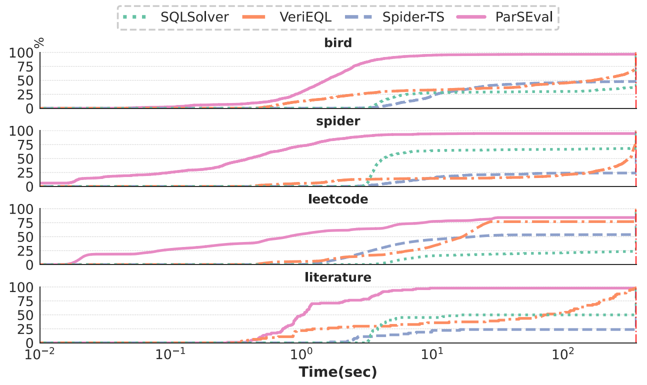


Figure 9: End-to-end performance distribution for all query pairs. A point (x, y) on the curve indicates that the approach finishes within x seconds for y percent of the query pairs. ParSEval can handle 95% query pairs within 8.5 seconds for BIRD, Spider, and Literature datasets.

- **Literature:** This dataset contains 64 query pairs from recent works on query equivalence [12, 34], also collected by [20].

5.2 Results of SQL Equivalence Checking

5.2.1 ParSEval Versatility and Effectiveness. We first evaluated how versatile our solution is and how effective the generated test cases are at finding non-equivalent query pairs. In Figure 8, we reported the ratio of non-equivalent and unsupported pairs from each approach for all four datasets. "Unsupported" indicates that the approach either fails to handle the query or is unable to return "equivalent" or "non-equivalent" within the timeout. For test-case-based approaches (Spider-TS and ParSEval), the outcome of a pair of queries will be 1) equivalent if both queries return non-empty and identical results on all generated databases, 2) not equivalent if two queries return different results on any of generated databases, or 3) unsupported if the approach can handle neither of the queries or both queries return empty results on all databases.

Regarding the support ratio (the right orange bar in Figure 8), ParSEval **supports most of the query pairs in all datasets** (97.98%, 96.71%, 92.69%, and 98.44%). Moreover, ParSEval can find **the most number of non-equivalent query pairs in all datasets** (the green bar in each plot in Figure 8) with an improvement over the best formal-verification-based baseline by at least 25.72, 20.51, 6.5, and 3.1 percentage points in four datasets, resp. On BIRD and Spider datasets, formal-verification-based approaches have a poor support ratio due to failing to handle complex SQL features, including `ORDER BY`, nested sub-queries, and `CAST` operations, which are common in current text-to-SQL benchmarks. In contrast, test-case-based approaches do not have to model the semantics of these operations and can more easily handle the queries.

Compared to the test instance generation baseline Spider-TS, ParSEval can also support more query pairs and disprove the equivalence of more query pairs. The improvement of ParSEval is clearer for human-crafted queries (Leetcode and Literature datasets), which is due to these datasets containing more SQL features not supported well in Spider-TS, such as attributes with data types like `Datetime` that require transformation, nested sub-queries, scalar

Table 3: Percentile distribution of the end-to-end running time for query pairs in the BIRD dataset. "Unsupp." indicates that the approach cannot handle the queries.

Approach	Mean	10th	25th	50th	75th	90th
SQLSolver	57.36	3.92	6.35	unsupp.	unsupp.	unsupp.
VeriEQL	96.52	0.82	2.65	169.21	unsupp.	unsupp.
Spider-TS	166.77	8.50	15.32	257.13	timeout	timeout
ParSEval	7.62	0.47	0.89	1.51	2.48	4.40

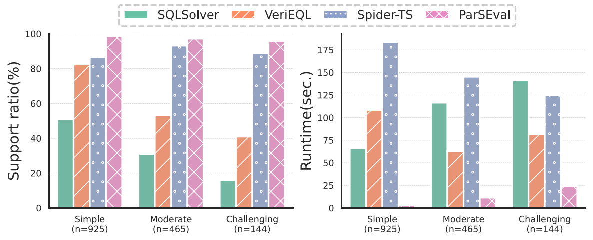


Figure 10: Breakdown of support ratio(left) and average runtime(right) across different difficulty levels (n is the number of query pairs for each difficulty level), BIRD dataset.

sub-queries, and arithmetic expressions. For example, consider Q_3 in Example 2, which has a scalar subquery in the `WHERE` clause, although Spider-TS could extract the join condition, the likelihood of generating test databases through fuzzing to cover the scalar subquery remains low.

5.2.2 Performance Evaluation. To evaluate the performance of ParSEval, we compared the end-to-end running time (from inputting the query pair to returning the equivalence decision) for each query pair of all methods. Figure 9 shows a Cumulative Distribution Function (CDF) of the running time. Although formal-verification-based approaches have a limited query support ratio, their performance is relatively better compared to Spider-TS because they directly verify the equivalence of symbolic representations using efficient SMT solvers (e.g., Z3), while fuzzing in Spider-TS needs to generate a large number of test instances. Note that ParSEval is able to disprove more query pairs than baselines, as shown in Figure 8, while achieving clearly better performance.

Moreover, Table 3 shows a detailed comparison of end-to-end running time on the BIRD dataset. Timeout indicates that the approach failed to prove or disprove the equivalence of the query pair within 360 seconds. Compared with other baselines, ParSEval can process at least 90% query pairs within 4.4 seconds, whereas Spider-TS takes 8.5 seconds to handle only 10% query pairs. Formal-verification-based approaches are relatively fast for the top 10%-25% queries, but they are much slower on average. On average, our approach is 21× faster than Spider-TS and 7.5× faster than SQLSolver.

5.2.3 ParSEval Scalability. We also studied how well ParSEval scales with the complexity of the input queries. Existing text-to-SQL benchmarks [25, 38] provide definitions of query difficulty, where difficulty is categorized into three levels based on the SQL query complexity, particularly the number and variety of keywords and functions used. More difficult natural language questions tend to involve more advanced SQL features and nested logic, resulting in longer and more complex queries. Here we reused the categorization from the original BIRD benchmark. Figure 10 presents a breakdown of the support ratio and average running time for the

BIRD dataset. The run time of ParSEval and SQLSolver increases with query difficulty, as expected, since the performance of encoding query semantics largely depends on the size of U-expressions and the number of different query execution paths in the query plan. Compared with baselines, the support ratio of ParSEval is the highest across all cases and is the most stable, validating the versatility of our approach. Spider-TS has the highest average running time on simple queries due to its lack of a termination strategy. Similarly, VeriEQL takes longer for simple queries because it attempts to prove query equivalence by enumerating possible tuples.

5.3 Absolute Performance and Case Study

To accurately evaluate the absolute performance of ParSEval, we manually annotated the BIRD dataset (979 non-eq pairs in all 1534 pairs) and computed the *recall* for each approach, defined as $\frac{\#\text{identified non-equivalent pairs}}{\#\text{human-labeled non-equivalent pairs}}$. The table below reports the results. Our method achieved both the highest recall (91.01%), indicating that ParSEval identified the **most number of non-equivalent query pairs**.

	SQLSolver	VeriEQL	Spider-TS	ParSEval
Recall	27.89%	46.17%	76.30%	91.01%

We also manually inspected query pairs that each approach failed to disprove and analyzed the reason for false positives.

Correlation and complex features. We observed that there are 26 pairs that Spider-TS was unable to generate any instances where two queries returned non-empty results due to the correlation of operators and complex query features. For example, given a query with correlated subquery, `select count(room_number) from ... where building = (select building from ... where dept_name='CS')`, which finds the total number of rooms in the building used by the CS department. Spider-TS will only consider the predicate of `dept_name = 'CS'`. In contrast, ParSEval will explicitly explore every execution path for the given query and generate data to cover the paths.

Projection errors. We also observed that formal verification-based approaches sometimes return queries as equivalent, while both Spider-TS and ParSEval correctly identify them as non-equivalent. This is because formal verification-based approaches will ignore projection when comparing two queries. For example, given the query pair <`SELECT Phone, Ext FROM ...`, `SELECT Phone FROM ...`>, it is obvious that they are different, yet formal verification-based approaches incorrectly output them as equivalent. Since such projection errors can be easily detected, we manually fixed these results.

NULL-related constraints. There were also 9 cases that ParSEval failed to disprove due to the NULL-related constraints for aggregate operators. For example, `SELECT COUNT(bond_id)` and `SELECT COUNT (bond_type)` are equivalent only if neither *bond_id* nor *bond_type* can be `NULL`. However, ParSEval failed to distinguish them because it did not handle varying numbers of NULL values. This limitation can be addressed by incorporating multiplicity constraints on `NULL` values during instance generation.

Quantitative analysis on SQL features. To further study the limitation of ParSEval and baseline methods beyond previously discussed failure cases, we summarized the occurrence of complex SQL features in the BIRD dataset and compared support rates across methods. Table 4 shows that ParSEval supports at least 78.95% of queries with these features, outperforming all existing methods for each feature. However, some operations, such as sequence string transformations (e.g., `AVG(CAST(SUBSTR(T2.LapTime, 1, INSTR(T2.LapTime, ':') - 1) AS INTEGER))`), remain challenging and may cause constraint solving to fail.

Table 4: Breakdown of support ratio (%) across query features, BIRD dataset. Spec. = Special operators. SubQ = Subquery.

	Case	Aggr.	Having	Distinct	Order by	Spec.	Typing	SubQ
#	117	113	19	237	313	62	154	113
VeriEQL	16.24	75.22	47.37	74.68	82.43	3.23	0.65	64.60
SQLSolver	31.62	14.16	26.32	41.77	18.21	0.00	29.22	19.47
Spider-TS	99.15	76.11	68.42	83.54	81.47	58.06	95.45	86.73
ParSEval	97.44	95.58	78.95	97.05	99.04	83.87	97.40	96.46

5.4 Ablation Study

We performed an ablation study to assess the impact of different coverage constraints on the running time and test data quality of ParSEval. We also compared ParSEval-Hybrid with the variant, i.e. ParSEval, that used pure symbolic execution.

Comparison of coverage constraints. We considered several variants for covering execution branches in the query plan: 1) Positive-only: covers only positive branch constraints. 2) ParSEval-OneNegative: covers positive branch constraints and one randomly chosen negative branch. 3) ParSEval: covers all positive and negative branch constraints. 4) ParSEval-Complete: covers different subsets of coverage constraints.

We reported the average running time and ratio of non-equivalent pairs found by each variant on the BIRD dataset in Table 5. The quality of test instances (as indicated by the ratio of NEQ pairs) increases with the number of coverage constraints. As expected, Positive-Only has the worst performance, producing around 10% false positives because it fails to identify the semantic differences between operators, such as left join and inner join. Compared to the ParSEval-OneNegative, ParSEval ensures that each coverage constraint is satisfied by at least one test instance, enabling it to distinguish minor variations in aggregate or set/bag operations. Interestingly (but as expected), ParSEval-Complete has the highest coverage at the cost of significantly increased running time, as it has to enumerate all subsets of the coverage constraints; however, it does not lead to more NEQ pairs identified. In summary, the ablation study indicates a trade-off between effectiveness and running time by enlarging the set of coverage constraints, and the relaxed completeness considered in ParSEval leads to the same test data quality as considering all combinations of the coverage constraints.

ParSEval-Sym vs. ParSEval-Hybrid. The table below shows the ratio of NEQ pairs, the number of timeout pairs, and the average running time for ParSEval-Sym and ParSEval-Hybrid using the same coverage strategy on the BIRD dataset.

	Support ratio	# of timeout	avg. time(sec.)
ParSEval-Sym	49%	222	9.29
ParSEval-Hybrid	98%	30	7.62

Due to the difficulty in modeling SQL functions and operators, such as `STRFTIME` and `CAST`, ParSEval-Sym could only handle 49% query pairs from the BIRD dev set. While uninterpreted functions in SMT solvers can model SQL functions, queries with outer joins, aggregates, and sorting involve unbounded summations in the U-expression, increasing SMT complexity and causing timeouts. In contrast, ParSEval-Hybrid reduces expression complexity by assigning concrete values to symbols, as shown in Section 4.3.

Table 5: Test data quality and average running time (sec.) for different sets of coverage constraints, BIRD dataset.

	Positive Only	ParSEval OneNegative	ParSEval	ParSEval Complete
% of disproved pairs	48%	53%	58%	58%
avg. running time (sec.)	4.92	6.11	7.62	20.83
avg. covered branches	1	1.85	4.22	13.52

5.5 Comparison with Text-to-SQL Benchmark

The above experiments demonstrate the superiority of ParSEval in checking query equivalence, which is already capable of facilitating efficient and automatic text-to-SQL evaluation. In this part, we evaluated the effectiveness of ParSEval in generating test suites, which takes the set of ground truth queries as input without seeing queries generated by text-to-SQL models. The test suite for each database schema is the union of all test suites generated by ParSEval for each ground truth query of the same schema. In particular, we compared the test suites generated by our approach and the database instances provided in the BIRD benchmark. The table below shows the number of non-equivalent pairs, the total time for evaluating one text-to-SQL model [16] on the dev set of the BIRD benchmark, and the total size of sqlite files of the test suites.

Method	Ratio of NEQ pairs	Total running time	Total disk space
ParSEval	59.45%	45.20 sec.	88 MB
BIRD test suite	50.33%	192.95 sec.	1.7 GB

Recall that to evaluate a text-to-SQL model on the current benchmarks, one needs to download the test suites and compare the results of generated queries with ground truth queries. ParSEval reduces the size of the database instances by about 20 times and speeds up the total evaluation time by 4.2 times.

False positives. For a detailed analysis, we manually examined the discrepancies in the equivalence evaluation results. The test suite by ParSEval introduces 41 false positives, mostly caused by wrong joins and special data types and functions like `JULIANDAY` in SQLite. For example, the ground truth query uses one table R, but the predicted query uses tables R and joins another table S; in this case, ParSEval will only create tuples in R and hence fail to distinguish the queries (when both queries are taken as inputParSEval is able to identify the inequivalence). The test suite in the BIRD benchmark introduces 180 false positives, typically due to SQL features like aggregation, string comparison, and distinct values. For example, the test suite in BIRD fails to distinguish the predicted query with a condition `WHERE T.amount > 100000` from the ground truth query with `GROUP BY T.id HAVING SUM(T.amount) > 100000`. Although test suites by ParSEval still introduce false positives, one can keep adding instances by running ParSEval on queries generated by text-to-SQL models to further improve the effectiveness of test suites.

6 RELATED WORK

Text-to-SQL translation and evaluation. Text-to-SQL allows users to query databases using natural language rather than writing SQL code. Over the years, various models, algorithms, and datasets [5, 9, 32, 38, 40] have advanced this field, leveraging deep learning from both database [24, 30] and NLP communities [23]. However, a universal evaluation framework for measuring semantic accuracy — correct SQL translations over total test queries — remains lacking. Recent studies use execution accuracy, but evaluating query equivalence at scale remains challenging due to the difficulty of preparing correct database instances.

Formal verification-based query equivalence evaluation. Formal verification-based query equivalence evaluation aims to determine whether two queries are semantically equivalent [15]. Despite being undecidable [2], significant effort has been devoted to automating this process [11, 15, 20, 35, 41]. Recent approaches leverage symbolic execution to develop practical solvers [12, 35, 42] that handle an expanding set of SQL queries. Cosette [12] models SQL semantics algebraically, applies rule-based rewriting, and checks query isomorphism. EQUITAS [41] and SPES [42] verify query equivalence by assessing containment relationships. QED [35] extends U-Expressions to Q-Expressions to model query semantics while additionally admitting an efficient encoding to SMT formulas. While being more efficient and feature-rich, these methods still have limitations in the query features they can handle.

Test case-based query equivalence evaluation. The testcase-based evaluation adapts the idea that two queries will yield the same results on any input database instance if they are semantically equivalent. This approach supports more query features but suffers from ad-hoc constraints and potential inaccuracies. RATest [28] proves two queries are inequivalent by generating the smallest counterexample where two queries return different results. XData [31] considers different types of common query errors, extracts constraints from SQL queries, and uses mutation techniques to kill as many query mutants as possible. Spider-TS [39] leverages fuzzing-based data generation to construct test databases for distinguishing two queries. Unlike ParSEval's guided approach, the input generation process of Spider-TS [39] does not use any information from past inputs but essentially creates new data with fuzzing from a prohibitively large input space.

Branch coverage in software testing. Branch coverage in software testing is used to ensure that every branch or path is tested [4, 19, 26, 33]. Coverage-based tools [3, 21, 29, 37] model individual program paths as logical constraints and employ symbolic execution to automatically generate arbitrary test inputs to maximize the branch coverage. Intuitively, higher test coverage reduces the likelihood of the software containing bugs and unforeseen errors. But these tools are not well-suited for query equivalence evaluation as they do not exploit the relative differences across multiple queries.

7 CONCLUSION AND FUTURE WORK

In this paper, we have studied the problem of test database generation for query equivalence evaluation by covering different branches. We present ParSEval, which considers all specific behaviors of each query operator in the logical query plan. We experimentally show the efficiency of our approach and the quality of the generated test cases on different datasets. The results show that ParSEval detected at least 8% more incorrect predictions than the previous fuzzy-testing-based test generation approach while being 21× faster. In future work, we plan to explore further optimizations for branch collection and integrate our approach with random data generation to support an even larger number of query pairs.

ACKNOWLEDGMENTS

This work was partly supported by NSERC Discovery Grants.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases*. Vol. 8. Addison-Wesley Reading.

[2] Serge Abiteboul and Victor Vianu. 1995. Computing with first-order logic. *Journal of computer and System Sciences* 50, 2 (1995), 309–335.

[3] S. B. Akers. 1978. Binary Decision Diagrams. *IEEE Trans. Comput.* 27, 6 (June 1978), 509–516. https://doi.org/10.1109/TC.1978.1675141

[4] Paul Ammann and Jeff Offutt. 2017. *Introduction to software testing*. Cambridge University Press.

[5] Ömer Aydın and Enis Karaarslan. 2022. OpenAI ChatGPT generated literature review: Digital twin in healthcare. *Available at SSRN 4308687* (2022).

[6] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442.

[7] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*. 221–230.

[8] Carsten Binnig, Donald Kossmann, Eric Lo, and M Tamer Özsu. 2007. QAGen: generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 341–352.

[9] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712* (2023).

[10] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 34–43.

[11] 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 (2018), 1482–1495.

[12] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *CIDR*.

[13] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: Proving query rewrites with univalent SQL semantics. *ACM SIGPLAN Notices* 52, 6 (2017), 510–524.

[14] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[15] Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. 2023. Proving Query Equivalence Using Linear Integer Arithmetic. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–26.

[16] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *Proceedings of the VLDB Endowment* 17, 5 (2024), 1132–1145.

[17] Amir Gilad, Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2022. Understanding queries by conditional instances. In *Proceedings of the 2022 International Conference on Management of Data*. 355–368.

[18] 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*. 31–40.

[19] Neelam Gupta, Aditya P Mathur, and Mary Lou Soffa. 2000. Generating test data for branch coverage. In *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*. IEEE, 219–227.

[20] Yang He, Pinhan Zhao, Xinyu Wang, and Yuepeng Wang. 2024. VeriEQL: Bounded Equivalence Verification for Complex SQL Queries with Integrity Constraints. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 1071–1099.

[21] Allen D Householder and Jonathan M Foote. 2012. Probability-based parameter selection for black-box fuzz testing. *Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2012-TN-019* (2012).

[22] Tomasz Imieliński and Witold Lipski Jr. 1984. Incomplete information in relational databases. *Journal of the ACM (JACM)* 31, 4 (1984), 761–791.

[23] 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)*. 963–973.

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

[25] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2024. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems* 36 (2024).

[26] Yashwant K Malaiya, Michael Naixin Li, James M Bieman, and Rick Karcich. 2002. Software reliability growth with test coverage. *IEEE Transactions on Reliability* 51, 4 (2002), 420–426.

[27] Hanze Meng, Zhengjie Miao, Amir Gilad, Sudeepa Roy, and Jun Yang. 2023. Characterizing and Verifying Queries Via CINSGEN. In *Companion of the 2023 International Conference on Management of Data*. 143–146.

[28] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining wrong queries using small examples. In *Proceedings of the 2019 International Conference on Management of Data*. 503–520.

[29] Roy P Pargas, Mary Jean Harrold, and Robert R Peck. 1999. Test-data generation using genetic algorithms. *Software testing, verification and reliability* 9, 4 (1999), 263–282.

[30] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R Mittal, and Fatma Özcan. 2016. ATHENA: an ontology-driven system for natural language querying over relational data stores. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1209–1220.

[31] Shetal Shah, S Sudarshan, Suhas Kajbaje, Sandeep Patidar, Bhanu Pratap Gupta, and Devang Vira. 2011. Generating test data for killing SQL mutants: A constraint-based approach. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 1175–1186.

[32] Yibo Sun, Duyu Tang, Nan Duan, Jianshu Ji, Guihong Cao, Xiaocheng Feng, Bing Qin, Ting Liu, and Ming Zhou. 2018. Semantic Parsing with Syntax-and Table-Aware SQL Generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 361–372.

[33] Javier Tuya, María José Suárez-Cabal, and Claudio De La Riva. 2010. Full predicate coverage for testing SQL database queries. *Software Testing, Verification and Reliability* 20, 3 (2010), 237–288.

[34] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2018. Speeding up symbolic reasoning for relational queries. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[35] Shuxian Wang, Sicheng Pan, and Alvin Cheung. 2024. QED: A Powerful Query Equivalence Decider for SQL. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3602–3614.

[36] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. Wetune: Automatic discovery and verification of query rewrite rules. In *Proceedings of the 2022 International Conference on Management of Data*. 94–107.

[37] Yaoxuan Wu, Ahmad Humayun, Muhammad Ali Gulzar, and Miryung Kim. 2024. Natural Symbolic Execution-Based Testing for Big Data Analytics. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2677–2700.

[38] 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*. 3911–3921.

[39] Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic Evaluation for Text-to-SQL with Distilled Test Suites. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 396–411.

[40] Victor Zhong, Mike Lewis, Sida I Wang, and Luke Zettlemoyer. 2020. Grounded Adaptation for Zero-shot Executable Semantic Parsing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 6869–6882.

[41] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. 2019. Automated verification of query equivalence using satisfiability modulo theories. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1276–1288.

[42] Qi Zhou, Joy Arulraj, Shamkant B Navathe, William Harris, and Jinpeng Wu. 2022. SPES: A symbolic approach to proving query equivalence under bag semantics. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2735–2748.

[43] Xuanhe Zhou, Guoliang Li, Jianming Wu, Jiesi Liu, Zhaoyan Sun, and Xinning Zhang. 2023. A learned query rewrite system. *Proceedings of the VLDB Endowment* 16, 12 (2023), 4110–4113.