Detecting Join Bugs in Database Engines via Join Implication Reasoning

ZHAOKUN XIANG, National University of Singapore, Singapore
SUYANG ZHONG, National University of Singapore, Singapore
MANUEL RIGGER, National University of Singapore, Singapore

Join is a fundamental operation in relational database management systems (RDBMSs), as it embodies core relational expressiveness of SQL, enabling users to query and analyze data across multiple tables. While approaches have been proposed specifically to find join bugs in RDBMSs, they are affected by scalability limitations. In this paper, we propose a novel, general, and effective technique for finding join logic bugs across all common join types, under arbitrary join predicates, and on join-supported RDBMSs, called *Join Implication Reasoning* (JIR). Our core insight is that the execution results of one or more join types allow us to infer the oracle of a semantic-related target join type. As an illustration, the execution results of *Inner Join* and *Anti Join* can be combined to infer the expected result of *Left Join* performed over the same left-hand side and right-hand side under the same join predicate. JIR validates joins by exploiting the join semantics of each individual DBMS itself for reasoning, and finds join logic bugs if the execution result of the target join fails to match the inferred oracle. We realized our approach and evaluated it on 11 extensively tested DBMSs: SQLite, MySQL, CockroachDB, ClickHouse, DuckDB, TiDB, MonetDB, Umbra, Dolt, CrateDB and PostgreSQL. Overall, JIR found 100 unique, previously unknown join bugs, of which 91 were fixed and 9 were verified by DBMS vendors, with 69 being join logic bugs. We expect that the generality and practicality of our approach will make it widely adoptable for understanding and testing *Join*.

CCS Concepts: • **Information systems** → **Relational database query languages**; **Database query processing**; • **Software and its engineering** → **Software verification and validation**.

Additional Key Words and Phrases: DBMSs testing, join logic bugs, join semantics, test oracle

ACM Reference Format:
Zhaokun Xiang, Suyang Zhong, and Manuel Rigger. 2026. Detecting Join Bugs in Database Engines via Join Implication Reasoning. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 159 (June 2026), 26 pages. https://doi.org/10.1145/3802036

1 Introduction

Modern relational database management systems (RDBMSs) distinguish themselves from non-relational database systems through their extensive support for complex joins. This enables them to perform well in business intelligence (BI) scenarios, such as complex data analysis and decision support, making them highly popular among data analysts. In order to process joins efficiently, query optimizers apply various sophisticated optimizations and select the least costly query plan [6]. Such optimizations include predicate push-down [38], join reordering [23], cost-based join method selection (*e.g.*, nested loops join, hash join, and sort-merge join), join elimination [14], and outer join transformation [16].

Authors' Contact Information: Zhaokun Xiang, xiangzk97@gmail.com, National University of Singapore, Singapore, Singapore; Suyang Zhong, suyang@u.nus.edu, National University of Singapore, Singapore, Singapore; Manuel Rigger, rigger@nus.edu.sg, National University of Singapore, Singapore, Singapore.

ACM 2836-6573/2026/6-ART159
https://doi.org/10.1145/3802036

The complexity and interactions of various join optimizations inevitably lead to join bugs. These bugs fall into two categories: crash bugs, which cause the system to crash, and more insidious ones—logic bugs. The latter do not cause system downtime, but rather silently return incorrect query results for particular data or query patterns, making them difficult to detect and debug. Even worse, join logic bugs may lead to incorrect results that propagate to a wide range of downstream systems and applications that depend on DBMSs, leading to incorrect analysis and decisions.

Two approaches have been proposed to automatically detect join logic bugs in RDBMSs: *Transformed Query Synthesis* (TQS) [44] and *Differential Query Plans* (DQP) [5]. TQS establishes the join ground-truth results by simulating join processing. DQP, in contrast, detects join logic bugs by checking discrepancies between results of different query plans enforced via query hints or system variables. However, both TQS and DQP suffer from fundamental limitations. Specifically, TQS is confined to testing equi-join. This is because the inherent limitation of TQS lies within its adoption of schema normalization methods [8, 33] to split a wide table into several sub-tables. Such normalizations are based on functional dependencies and introduce primary key (PK)-foreign key (FK) relationships between sub-tables, so the join ground-truth results can be recovered from the wide table through PK-FK join among sub-tables. Therefore, TQS only supports the generation and validation of PK-FK equi-joins.[1] However, real-world analytical workloads feature diverse join patterns, including non-equi-joins and many-to-many joins between attributes without PK-FK relationships [9]. Limiting the scope to the PK-FK equi-join overlooks bugs that arise from non-equi-join conditions, and we have found 25 such logic bugs in our evaluation (see Section 5.3). DQP cannot be applied to DBMSs that lack support for query hints, such as SQLite, DuckDB, or MonetDB, whereas our method found a total of 41 join logic bugs on these three systems. More importantly, both TQS and DQP scale poorly with complex join features, such as joins with subqueries. Conceptually, it is unclear how TQS could derive ground-truth results for joins with subqueries; although DQP could, in principle, also be applied to complex features such as subqueries, its effectiveness is hindered by divergent query hint support across DBMSs. For example, TiDB provides the *no_decorrelate* hint to disable decorrelation for subqueries, whereas MySQL lacks such functionality. Therefore, effectively orchestrating divergent query hints among DBMSs for complex queries faces scalability issues in practice.

To scale the join logic bug detection across diverse join types and patterns with arbitrary join predicates, while also supporting systems that lack query hint instructions, it is imperative to have a general join oracle. However, developing a general join oracle is a significant challenge. This challenge stems from the diversity and complexity of join operations, which can join divergent data sources such as base tables, views, and subqueries under various join conditions beyond simple equality. Furthermore, different DBMSs implement varying semantics for operators and expressions, making it difficult to scale a "one-size-fits-all" approach like TQS, which simulates join results, to different DBMSs.

To overcome this challenge, we propose a novel join logical bug detection method called *Join Implication Reasoning* (JIR). We leverage the join semantics of each individual DBMS for reasoning. The core idea is, assuming the execution results of one or more join types are correct and leveraging these results, to infer the expected result of the logically related target join type. If the actual execution result of the target join mismatches the expected result inferred by JIR, join operators involved in reasoning are affected by join logical bugs. In this way, JIR uses the DBMS under test itself as a part of the oracle, avoiding the reliance on sophisticated emulation of join ground-truth results or query hints.

[1]Section 3.4 of the TQS paper highlights this limitation: "Since we stick to the primary-foreign key joins [...]"

Listing 1. A multi-join bug that was latent in SQLite for over five years

```
CREATE TABLE t0 (c0 INT);
CREATE TABLE t1 (c0 INT);
INSERT INTO t0 VALUES (0);
INSERT INTO t1 VALUES (1);
-- Inner Join Result
SELECT t0.c0
FROM (SELECT TOTAL(0) OVER () AS c0) as sub
LEFT JOIN t0 ON ((CASE 1 WHEN 1 THEN sub.c0 END)
LIKE sub.c0) INNER JOIN t1 ON (sub.c0 == false);
-- {NULL} 🐞 -- {0} ✔

-- Anti Join Result
SELECT t0.c0
FROM (SELECT TOTAL(0) OVER () AS c0) as sub
LEFT JOIN t0 ON ((CASE 1 WHEN 1 THEN sub.c0 END) LIKE sub.c0) WHERE NOT EXISTS
    (SELECT 1 FROM t1 WHERE (sub.c0 == false));
-- Empty Result ✔

-- Left Join Result
SELECT t0.c0
FROM (SELECT TOTAL(0) OVER () AS c0) as sub
LEFT JOIN t0 ON ((CASE 1 WHEN 1 THEN sub.c0 END) LIKE sub.c0) LEFT JOIN t1 ON
    (sub.c0 == false);
-- {0} ✔
```

As a motivating example, Listing 1 presents an over 5-year-old SQLite bug,[2] which we found using JIR. Intuitively, according to the semantics of `LEFT JOIN`, given a left relation L, a right relation R, and a join predicate P, when executing `L LEFT JOIN R ON P`, each tuple l from L is processed as follows: (1) If one or more tuples r in R match l based on the predicate P, all qualifying (l, r) pairs are included in the result of the `LEFT JOIN`. This part is semantically equivalent to `L INNER JOIN R ON P`. (2) If no tuple r in R matches l, l is nevertheless preserved in the left part of a result tuple (l, r'). Such preservation semantics for tuple l is the same as that in `L ANTI JOIN R ON P`. The right part of the left join result tuple, r', will be filled in with NULL for every R's attributes. Therefore, `L LEFT JOIN R ON P` is equivalent to the union of `L INNER JOIN R ON P` and `L ANTI JOIN R ON P`, where each tuple l from the `ANTI JOIN` result is extended to (l, r'), and all attributes of r' are explicitly filled with NULL to construct the right-hand side consistent with the `LEFT JOIN` result. According to this semantic relationship, in this example, if `LEFT JOIN` produces the result set $\{(0)\}$ and `ANTI JOIN` produces an empty result, the correct `INNER JOIN` result should be the $\{(0)\}$. However, the execution result of `INNER JOIN` is $\{(NULL)\}$, violating the expected semantic relationship, revealing a join logic bug in SQLite. Such a semantic-conflicting-driven strategy circumvents the complexity and potential risks of implementing a correct join simulator like TQS. Additionally, this bug is a non-equi-join logic bug, which TQS conceptually fails to derive the ground truth result for, and SQLite lacks support for query hints, making DQP unable to detect it.

We implemented our approach on SQLancer, a DBMS testing tool, and evaluated it on 11 DBMSs, including SQLite, MySQL, CockroachDB, ClickHouse, DuckDB, TiDB, MonetDB, Umbra, Dolt, CrateDB and PostgreSQL, finding 100 unique, previously unknown join bugs, of which 91 were

[2]https://www.sqlite.org/forum/forumpost/35bf50ea01

fixed, 9 were verified by DBMS vendors, with 69 being join logic bugs. In addition, we received appreciative feedback from DBMS developers. For example, Richard Hipp, the creator of SQLite, showed great interest when we found a bug that extended the reach of a left join bug submitted more than 5 years ago.[3] He also appreciated that we discovered and reported an elusive bug before the official release of a join optimization rule.[4]

Overall, we make the following contributions.

- At the conceptual level, we propose the idea of Join Implication Reasoning (JIR), a novel, general, and effective technique designed for finding join logic bugs in DBMSs, which infers the oracle of a target join type by leveraging execution results of logically related join types.
- At the technical level, we propose realizations of JIR to test `INNER`, `LEFT`, `RIGHT`, `FULL`, `CROSS`, `SEMI`, `ANTI`, `NATURAL`, and `LATERAL JOIN`.
- At the practical level, we realized our approach on SQLancer, and evaluated it on SQLite, MySQL, CockroachDB, ClickHouse, DuckDB, TiDB, MonetDB, Umbra, Dolt, CrateDB, and PostgreSQL. We successfully detected 100 unique, previously unknown join bugs, of which 91 were fixed and 69 bugs were join logic bugs.

2 Background

In this section, we establish the background about SQL predicates and joins.

Predicate. SQL relies on predicates, which are expressions that return `TRUE`, `FALSE`, or `NULL` when applied to input rows. Predicates serve as filters or matching conditions and appear in a wide range of query clauses, including filtering rows with `WHERE`, specifying grouping with `GROUP BY`, defining grouping constraints in `HAVING`, or controlling sorting via `ORDER BY`. When predicates are used in `JOIN ON`, they determine which tuples from the joining relations are considered a match; in this context, they are commonly referred to as join conditions.

Join. `JOIN` is one of the most fundamental operations in relational database systems, combining tuples from two relations based on a specified predicate as the join condition [28]. In standard SQL, `JOIN` is classified into different types depending on how tuples from relations are combined. Therefore, different join types have different semantics for tuple matching and preservation. Let L and R denote the left-hand side (LHS) and right-hand side (RHS), respectively. Figure 1 summarizes the fundamental join types and their semantics. Let P denote the join predicate, on which all join types requiring join conditions are performed, enabling 3 and 4 of LHS to match A and B of RHS, respectively. In addition to above join types, next we will introduce two other specialized join types: `NATURAL JOIN` and `LATERAL JOIN`. As detailed in the Section 3.2, JIR supports all these join types.

`NATURAL JOIN` is a special equi-join, which joins on the conjunction of all attributes with the same name between two relations. It then performs a projection, which eliminates duplicate common attributes. By default, `NATURAL JOIN` performs a `NATURAL INNER JOIN`, and `NATURAL` clause can also be combined with `LEFT JOIN`, `RIGHT JOIN`, and `FULL JOIN` according to dialects of different DBMSs.

`LATERAL JOIN` is a dependent join [31], in which the RHS is dependent on the LHS. In such a join, the RHS can reference columns from the LHS, and it is evaluated once for every tuple of the LHS. It is such a left-to-right evaluation dependency that makes `LATERAL JOIN` differ from conventional join processing, where both inputs are evaluated independently before the join. Consequently, such left-hand side evaluation prioritization makes `LATERAL JOIN` only applicable to `INNER JOIN` and `LEFT JOIN`.

[3] https://sqlite.org/src/info/623eff57e76d45f6
[4] https://sqlite.org/src/info/e33da6d5dc964db8

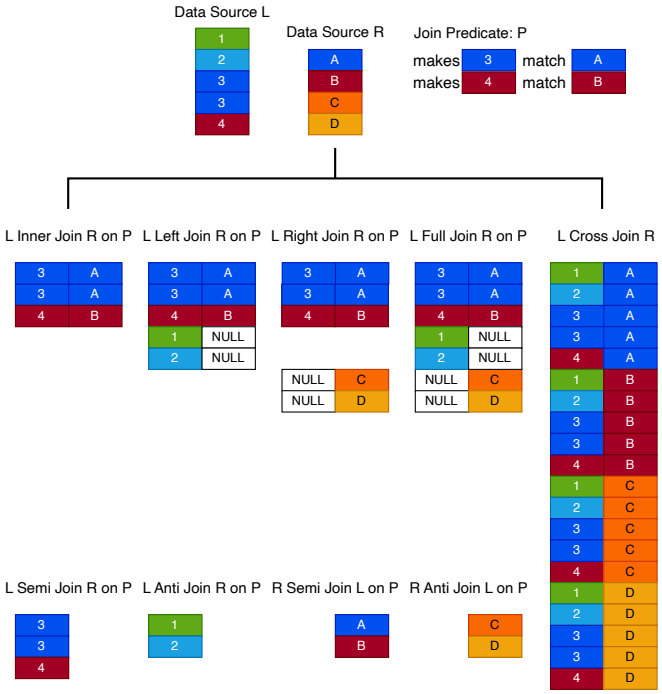

Fig. 1. Tuple matching and preservation semantics of join types

3 Join Implication Reasoning

In this section, we provide a conceptual overview of JIR, describe the reasoning rules for join oracle generation, and discuss the main characteristics of our method.

3.1 Overview

The core idea behind *Join Implication Reasoning* (JIR) is that, given one or more join execution results, derived from different join types under the same predicate P, performed over the same data sources L and R, within the same DBMS, to infer the oracle of a semantically related target join type in that DBMS. The following formula represents the reasoning process of JIR, which derives the expected semantics of the target join type from the observed execution results of related joins.

$$\mathrm{JIR}\Big(\mathrm{DB}(L \bowtie_1^P R),\ \mathrm{DB}(L \bowtie_2^P R),\ \ldots,\ \mathrm{DB}(L \bowtie_n^P R)\Big) = \mathcal{O}(L \bowtie_{\text{target}}^P R)$$

The $\mathcal{O}(L \bowtie_{\text{target}}^P R)$ is the inferred oracle for the target join; any discrepancy observed in the DBMS indicates a logic bug. Additionally, L and R denote arbitrary data sources—such as base tables, views, subqueries, table-generation functions, common table expressions (CTE), or intermediate join results—as long as they are joinable within a given DBMS. JIR() denotes an implication reasoning mechanism that leverages the execution results DB() of joins $\bowtie_1^P, \ldots, \bowtie_n^P$ to infer the expected result of the target join $\bowtie_{\text{target}}^P$, assuming all joins share the same LHS data source L, RHS data source R and join predicate P. Conceptually, such an implication reasoning mechanism JIR() can

be compositions of relational algebra operations, such as `UNION`, `INTERSECT`, and `EXCEPT`, over the results of semantically related join types $\bowtie_1, \bowtie_2, \ldots, \bowtie_n$, supported by the DBMS.

The foundation of JIR lies in a general principle: If two join types $\bowtie_1$ and $\bowtie_2$ follow a certain implication relationship $\mathcal{R}$, their corresponding execution results, based on the same join predicate P, performed over the same pair of joinable data sources L and R, should also conform to such a semantic relationship. This principle can be formally expressed by the following formula.

$$\bowtie_1 \ \mathcal{R} \ \bowtie_2 \rightarrow \mathrm{DB}\big((L \ \bowtie_1 \ R) \ \mathrm{On}\ P\big) \ \mathcal{R} \ \mathrm{DB}\big((L \ \bowtie_2 \ R) \ \mathrm{On}\ P\big)$$

Specifically, $\mathcal{R}$ denotes an implication relationship between them, such as equivalence ($\equiv$), containment ($\subseteq$), or disjointness ($\perp$). For example, suppose $L(id) = \{(1), (2), (2)\}$ and $R(id) = \{(2)\}$ with predicate P as $L.id = R.id$. Executing an `INNER JOIN` as $\bowtie_1$ yields $\{(2, 2), (2, 2)\}$, whereas a `LEFT JOIN` serving as $\bowtie_2$ yields $\{(1, \mathsf{NULL}), (2, 2), (2, 2)\}$. Intuitively, the execution result of the `INNER JOIN` is contained in the result of the `LEFT JOIN`, illustrating the semantic relationship between $\bowtie_1$ and $\bowtie_2$.

3.2 Implication Reasoning Rules

JIR is extensible and admits a broad class of implication reasoning rules, as long as they are sound in terms of join semantics. In this work, we realize six rules that are both widely applicable and have proven effective in finding join logic bugs. The goal of our proposed six rules is to support all commonly supported join types, but to avoid any overlap; removing any rule would result in bugs specific to the corresponding join type being missed. The six rules allow testing of all commonly supported join types, including `INNER`, `LEFT`, `RIGHT`, `FULL`, `CROSS`, `NATURAL`, `SEMI`, and `ANTI`, across arbitrary joinable data sources (including `LATERAL` reference) and under arbitrary join predicates.

Figure 1 visualizes the semantic relationships between a wide range of join types applied to the same data sources L and R under the same predicate. These examples make the inclusion and compositional relationships among join types explicit and provide visual intuition for the six implication reasoning rules we realize. We present a representative proof for one of our rules in Section 3.3.

Left Join Decomposition. The expected result of a `LEFT JOIN` can be inferred by composing the execution result of an `INNER JOIN` and an `ANTI JOIN` with appropriate NULL-padding:

$$L \bowtie^{\mathrm{On\ P}}_{\mathrm{LEFT}} R = (L \bowtie^{\mathrm{On\ P}}_{\mathrm{INNER}} R) \cup \mathcal{N}(L \bowtie^{\mathrm{On\ P}}_{\mathrm{ANTI}} R)$$

Here, $\mathcal{N}$ represents NULL-padding of R's attributes for each unmatched tuple from L. This ensures that the combined result corresponds to the semantics of `LEFT JOIN`.

Left/Right Join Symmetry. `RIGHT JOIN` is semantically equivalent to `LEFT JOIN` with the LHS and RHS swapped, and *vice versa*. When projecting all columns implicitly (via `SELECT` $*$), column reordering is required to preserve the original attribute order:

$$L \bowtie^{\mathrm{On\ P}}_{\mathrm{RIGHT}} R = \pi_{L,R}(R \bowtie^{\mathrm{On\ P}}_{\mathrm{LEFT}} L)$$

Here, $\pi_{L,R}$ denotes a reordering that rearranges columns to match the left-to-right order of attributes from L and R. In addition, this rule allows reasoning about `RIGHT JOIN` through an equivalent `LEFT JOIN`, and thus left join decomposition can be applied.

Semi/Anti Join Complement. In the context of `SEMI JOIN` and `ANTI JOIN`, each tuple in a data source L can be attributed to one of two disjoint subsets based on a join predicate P: a tuple set in which each tuple matches at least one tuple in data source R under predicate P, and a tuple set in which each tuple does not match any tuple in data source R:

$$L = (L \bowtie^{\mathrm{On\ P}}_{\mathrm{SEMI}} R) \cup (L \bowtie^{\mathrm{On\ P}}_{\mathrm{ANTI}} R)$$

Such two tuple sets correspond to **SEMI JOIN** and **ANTI JOIN**, respectively. The combination of these two complementary results restores exactly the original result set of L, even if there are duplicate tuples in L.

Full Join Decomposition. `L` **FULL JOIN** `R` **ON** `P` can be expressed as the union of three disjoint joins: `L` **INNER JOIN** `R` **ON** `P`, `L` **ANTI JOIN** `R` **ON** `P` (NULL-padded on the R side), and `R` **ANTI JOIN** `L` **ON** `P` (NULL-padded on the L side):

$$L \bowtie^{\text{On P}}_{\text{FULL}} R = (L \bowtie^{\text{On P}}_{\text{INNER}} R) \cup \mathcal{N}_R(L \bowtie^{\text{On P}}_{\text{ANTI}} R) \cup \pi_{L,R}(\mathcal{N}_L(R \bowtie^{\text{On P}}_{\text{ANTI}} L))$$

Here, $\pi_{L,R}$ denotes rearrangement of columns for matching the left-to-right order of attributes from L and R. $\mathcal{N}_R$ and $\mathcal{N}_L$ denote the NULL-padding on the attributes of R and L, respectively. Like left join decomposition, this construction ensures that all matched and unmatched tuples from both L and R are retained in accordance to the semantics of **FULL JOIN**.

Cross Join Equivalence. Assuming both input relations are non-empty, **CROSS JOIN** can be expressed as **INNER JOIN**, **LEFT JOIN**, **RIGHT JOIN**, or **FULL JOIN**, all with the join condition explicitly set to **TRUE**:

$$\begin{aligned} L \bowtie_{\text{CROSS}} R &= L \bowtie^{\text{On TRUE}}_{\text{INNER}} R = L \bowtie^{\text{On TRUE}}_{\text{LEFT}} R \\ &= L \bowtie^{\text{On TRUE}}_{\text{RIGHT}} R = L \bowtie^{\text{On TRUE}}_{\text{FULL}} R \end{aligned}$$

Such equivalence stems from the semantics of **CROSS JOIN** and the role of **JOIN ON** predicate. Assuming that both the LHS and RHS are non-empty, **CROSS JOIN** is defined to generate the Cartesian product of two relations, combining each row of the first relation with each row of the second relation. For join types requiring predicates, when the join predicate is always true, it becomes a condition that makes any tuple from the LHS match with any tuple from the RHS. In this context, **NULL**-padding for unmatched rows in outer join types never occurs. Therefore, when both input relations are non-empty, and the **ON** predicate is set to true, **INNER JOIN**, **LEFT JOIN**, **RIGHT JOIN**, and **FULL JOIN** all fetch every row combination, producing the same Cartesian product as a **CROSS JOIN** that shares the same LHS and RHS.

Natural Join Condition Explication. **NATURAL JOIN** can be regarded as an implicit equal inner join whose condition is the conjunctive equality of all attributes that share the same name and compatible data types in both input relations. Formally, let X be the set of attributes common to both L and R, where each $x \in X$ denotes such an attribute, and we use the notation $\equiv_{\text{rows}}$ to denote equivalence under row-matching semantics. Since **NATURAL JOIN** removes attributes that share duplicated names, while **INNER JOIN** does not, the two may differ in output schemas, but still produce result sets with identical row counts.

$$L \bowtie_{\text{NATURAL}} R \equiv_{\text{rows}} L \bowtie^{\bigwedge_{x \in X} L.x = R.x}_{\text{INNER}} R$$

Similarly, outer variants of natural joins can be explicated in the same manner. **NATURAL LEFT JOIN**, **NATURAL RIGHT JOIN**, and **NATURAL FULL JOIN** can be reinterpreted by their respective outer joins with the conjunctive equality predicate constructed from all shared attributes. Importantly, this rule makes join conditions of **NATURAL JOIN** explicit according to its semantics, and thus previously proposed reasoning rules can be applied to **NATURAL JOIN**.

3.3 Proofs

In this section, we prove the correctness of the *Left Join Decomposition* rule. As the proposed rules are grounded in the semantics of relational joins, the proof procedures are repetitive and can be obtained by repeatedly applying the formal definitions of join types. For brevity, we only include

the proof of the *Left Join Decomposition* rule in the main body of the paper, and defer the complete proofs of all rules to the artifact repository of the paper.[5]

Notation and definitions. Let L and R be relations and P an arbitrary join predicate. According to the definition, `INNER JOIN` can be represented as $L \bowtie^{\text{On } P}_{\text{INNER}} R = \{ \ell \circ r \mid \ell \in L,\ r \in R,\ P(\ell, r) \}$, and `ANTI JOIN` can be represented as $L \bowtie^{\text{On } P}_{\text{ANTI}} R = \{ \ell \mid \ell \in L,\ \neg(\exists r \in R :\ P(\ell, r)) \}$, where $\circ$ denotes tuple concatenation, *i.e.*, it constructs a tuple by concatenating the attributes of ℓ and r. Then, we let $\mathcal{N}(\cdot)$ denote the null-padding operator that converts each $\ell \in L$ into $\ell \circ t_{\text{null}_R}$, where t_{null_R} is the tuple over Schema(R) whose attributes are all NULL. Thus,

$$\mathcal{N}(L \bowtie^{\text{On } P}_{\text{ANTI}} R) = \{ \ell \circ t_{\text{null}_R} \mid \ell \in L, \neg(\exists r \in R :\ P(\ell, r)) \}.$$

Then, the left outer join can be denoted by $BAG_1 \coloneqq L \bowtie^{\text{On } P}_{\text{LEFT}} R$ and the union of inner join and anti join can be denoted by

$$BAG_2 \coloneqq (L \bowtie^{\text{On } P}_{\text{INNER}} R) \cup \mathcal{N}(L \bowtie^{\text{On } P}_{\text{ANTI}} R).$$

Proof. We need to prove that $BAG_1 \subseteq BAG_2$ and $BAG_2 \subseteq BAG_1$.

(1) $BAG_1 \subseteq BAG_2$. Let $t \in BAG_1$. By the standard definition of left outer join, each $t \in BAG_1$ is either

- Of the form $\ell \circ r$ with $\ell \in L,\ r \in R$ and $P(\ell, r)$ (a matching tuple), or
- of the form $\ell \circ t_{\text{null}_R}$ with $\ell \in L$ and $\neg(\exists r \in R :\ P(\ell, r))$ (an unmatched left tuple padded with NULLs).

In the first case, $t \in L \bowtie^{\text{On } P}_{\text{INNER}} R \subseteq BAG_2$. In the second case, $\ell \in L \bowtie^{\text{On } P}_{\text{ANTI}} R$, hence $t = \ell \circ t_{\text{null}_R} \in \mathcal{N}(L \bowtie^{\text{On } P}_{\text{ANTI}} R) \subseteq BAG_2$. Thus, every $t \in BAG_1$ lies in BAG_2, proving $BAG_1 \subseteq BAG_2$.

(2) $BAG_2 \subseteq BAG_1$. Let $t \in BAG_2$. Then t belongs to one of the two components of the union.

- If $t \in L \bowtie^{\text{On } P}_{\text{INNER}} R$, then $t = \ell \circ r$ with any ℓ and r satisfying $P(\ell, r)$; such tuples are included in the left outer join by definition, so $t \in BAG_1$.
- If $t \in \mathcal{N}(L \bowtie^{\text{On } P}_{\text{ANTI}} R)$, then $t = \ell \circ t_{\text{null}_R}$ for every $\ell \in L$ with $\neg(\exists r \in R :\ P(\ell, r))$. By the definition of the left outer join, unmatched left tuples are included and padded with NULLs; hence $t \in BAG_1$.

Hence, every $t \in BAG_2$ lies in BAG_1, which proves $BAG_2 \subseteq BAG_1$.

Combining (1) and (2) we obtain $BAG_1 = BAG_2$, i.e.

$$L \bowtie^{\text{On } P}_{\text{LEFT}} R = (L \bowtie^{\text{On } P}_{\text{INNER}} R) \cup \mathcal{N}(L \bowtie^{\text{On } P}_{\text{ANTI}} R).$$

3.4 Properties

Generality. JIR is a general approach for detecting join logic bugs. Its core idea is grounded in join semantics: it infers the expected result of a target join type by leveraging the actual execution results of other logically related join types under the same predicate. This design enables JIR to detect join logic bugs without relying on complex simulations of join ground-truth results or query hints, thereby ensuring generality across all common join types, under arbitrary join predicates, over arbitrary joinable data sources, and on arbitrary join-enabled relational DBMSs.

Another manifestation of JIR's generality is that our realizations are fundamentally designed to reason about joins under more general bag semantics rather than being limited to set semantics. Join semantics belong to query semantics, which are generally classified into set semantics and bag semantics [2]. Set semantics can be regarded as a special case of bag semantics, because duplicates in relations and query results are not considered under set semantics, while bag semantics retains

[5]https://doi.org/10.5281/zenodo.19000645

them, treating relations and results as multisets [22]. In practice, DBMSs typically default to the more general bag semantics [10]. However, TQS only detects join logic bugs in the context of set semantics. This is because it removes duplicates based on the primary key of results,[6] which will miss some bugs containing duplicates. Theoretically, JIR's capability to detect join logic bugs under bag semantics subsumes that of set semantics. This is because the reasoning rules we propose are inherently designed for the more general bag semantics, which can handle duplicate tuples in inputs and results. Specifically, we shift the granularity of join predicate evaluation to every possible join type and decompose more permissive join types into finer-grained restrictive join types. Results of such restrictive join types are then combined via `UNION ALL`, which will retain all duplicated rows and conform to bag semantics. Hence, JIR is general in testing `JOIN`.

Effectiveness. The effectiveness of JIR lies in detecting join logic bugs by identifying semantic conflicts in execution results across different join types under the same join conditions. DBMSs apply various optimization strategies for different join types. For instance, join predicate push-down [38] and traditional join reordering optimizations apply only to `INNER JOIN`, while outer join elimination [14] applies only to outer joins. Furthermore, different join types have diverse functionalities and execution paths. For instance, `NATURAL JOIN` automatically constructs equi-join conditions involving columns that share the same name and have compatible data types, and `NULL` padding semantics occur only in outer joins, which can potentially be prone to bugs. As a result, certain bugs in join optimizations or implementations may manifest only in relevant join types. By performing a cross-join-type semantic consistency check, JIR directly targets join semantics and reveals latent logic bugs in the optimization or execution of various join types.

4 Implementation

Our proposed six reasoning rule can be realized in any DBMS testing tool that supports, or can be extended to support, common relational `JOIN ON` clauses with expressions as join predicates. We first implemented JIR in SQLancer,[7] which is a logic bug detection tool for DBMSs and has integrated several test oracles [5, 35–37, 40, 46] for finding logic bugs. SQLancer relies on manually implemented generators to construct the database state and queries tailored to each DBMS under test. These generators incorporate a wide range of features for producing database states and queries with complex expressions. Among the 11 DBMSs we chose, SQLancer lacked support for Umbra and MonetDB, but we implemented generators for them. Apart from these generators, the implementation for JIR in SQLancer comprises approximately 400 lines of code for each DBMS. Additionally, we implemented JIR in SQLancer++ [49], which is a more recent DBMS testing tool and employs an adaptive generator that applies to any relational DBMS. JIR helped SQLancer++ to find additional join logic bugs in DBMSs (*e.g.*, CrateDB and Dolt), for which SQLancer lacks support. The workflow of JIR is shown in Figure 2, and we next describe the key implementation details for each step.

In step ①, we randomly generate a join query that is represented as an Abstract Syntax Tree (AST) style join tree, as detailed in Section 4. Green leaf nodes denote arbitrary joinable data sources, such as base tables or subqueries, and internal nodes denote join operators. All join predicates are randomly generated boolean expressions. Since the generated join query may contain multiple join types, we treat the join type of the root node as the overall join type of the query. The root node acts as a join-type placeholder, which will later be replaced with different join types.

In step ②, we instantiate the join-type placeholder with all join types supported by the target DBMS to generate transformed join queries. For join types such as `SEMI JOIN` and `ANTI JOIN`,

[6] Section 3.4 of the TQS paper explains: "Duplicated tuples are removed based on the new primary key of the tuples [...]"
[7] https://github.com/sqlancer/sqlancer

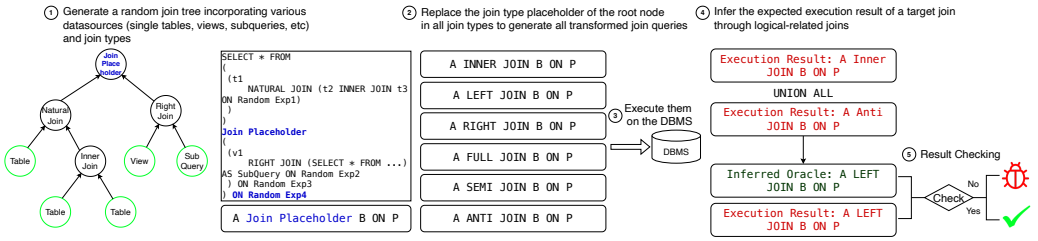

Fig. 2. The implementation architecture of JIR

we transform them using `WHERE EXISTS` and `WHERE NOT EXISTS`, respectively. If a DBMS natively supports the `SEMI JOIN` and `ANTI JOIN` clause, such as for DuckDB, we execute both the keyword-based and the `EXISTS`-based transformations.

In step ③, we execute all generated queries on the target DBMS to obtain their execution results. In step ④, based on the reasoning rules introduced in Section 3.2, we infer the expected result of a target join type from the execution results of semantically related join types. For example, the result of a `LEFT JOIN` can be inferred from the results of `INNER JOIN` and `ANTI JOIN`.

Finally, in step ⑤, we compare the inferred oracle of the target join with its actual execution result. Any discrepancy indicates a violation of the semantic implication among the involved join types used in the reasoning process, revealing a join logic bug among these join types.

Join generation. While SQLancer already generated joins for multiple DBMSs, the join generation mechanism was rudimentary. For example, it lacked support for joins with subqueries. To exercise more advanced join optimizations, we advanced SQLancer's join generation. A recursive join tree skeleton consists of leaf nodes and join nodes. Every leaf node can be a base table, view, or subquery. Join nodes are recursively built by combining leaf nodes or previously constructed join nodes, forming the overall join tree. When the RHS of a join node is a subquery, for DBMSs supporting `LATERAL JOIN`, the generator randomly rewrites it into a lateral subquery by referencing attributes from the LHS. Finally, join types are randomly selected for every join node, and join predicates are randomly generated via SQLancer's AST expression generator based on the visible columns of both LHS and RHS of join nodes.

Previous works [13, 41] generated join graphs [30] by defining each table as a vertex and each join predicate that connects exactly two tables as an edge. However, our join patterns incorporates advanced features such as joins with subqueries, and join conditions are random expressions. It extends beyond the scope of simple equi-joins between single tables, which is the only scenario considered by the definition of join graphs. In addition, all join graphs can be alternatively represented as join trees, since any complex join graph can be constructed by a series of binary join steps. Generating complex joins is not a conceptual contribution of this paper; currently, both join patterns and predicates are generated randomly.

5 Evaluation

We evaluated our approach and sought to answer the following questions to demonstrate the effectiveness of our approach.

- ***Q1*** Can our approach find join bugs in widely used and well-tested DBMSs (Section 5.2)?
- ***Q2*** How diverse are the join optimization bugs found by our approach (Section 5.3 and Section 5.4)?
- ***Q3*** Can JIR detect join logic bugs beyond the reach of existing methods (Section 5.5)?

5.1 Experimental Setup

Selected DBMSs. To evaluate the effectiveness and generality of our approach in detecting join optimization bugs, we selected 11 DBMSs: SQLite, MySQL, CockroachDB, ClickHouse, DuckDB, TiDB, MonetDB, Umbra, Dolt, CrateDB, and PostgreSQL. Our selection was motivated by three factors: popularity, variety of join support, and prior testing coverage.

In terms of popularity, these systems are widely used in academic or industrial settings. MySQL, PostgreSQL, and SQLite are among the most widely adopted and popular database management systems. According to the DB engine rankings,[8] they consistently rank among the top in terms of global usage: MySQL and PostgreSQL are client-server DBMSs, while SQLite is an embedded database engine deployed in many devices or integrated with countless mobile applications. CockroachDB [42] and TiDB [17] are distributed relational database systems, highly popular on GitHub. DuckDB [34], MonetDB [29], and Umbra [32] originate from academic research groups and have gained broad adoption through commercialization.

Regarding the support of the selected DBMSs for joins and their optimizations, different systems have varying degrees of support for join types. We found that foundational join types are universally supported, as all our 11 selected systems provide support for `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `NATURAL JOIN`, and `CROSS JOIN`. Additionally, all of them can express `SEMI JOIN` and `ANTI JOIN` using subqueries such as `EXISTS` or `IN`. However, we observed some inconsistencies across these systems. For instance, `FULL JOIN` is absent in MySQL and TiDB. Furthermore, while all systems can express `SEMI JOIN` and `ANTI JOIN` by `EXISTS/IN` and `NOT EXISTS/IN`, respectively, DuckDB is the only one to provide direct syntax for `SEMI JOIN` and `ANTI JOIN`. `LATERAL JOIN` is supported only by MySQL, CockroachDB, DuckDB, MonetDB, Umbra, PostgreSQL, and Dolt. To our knowledge, no existing work supports the generation and validation of such join types, and thus finding bugs triggered only by `LATERAL JOIN` can be evidence of our approach's effectiveness; as subsequently elaborated, we found 8, 2, and 2 such bugs in DuckDB, MonetDB, and Umbra, respectively.

For prior testing coverage, all the DBMSs we selected were extensively tested by a variety of logic bug detecting methods [5, 15, 20, 35–37, 40, 44, 46] or crash bug detecting methods [12, 18, 24, 25, 47], as well as testing platforms [3, 26, 49], indicating that newly found bugs can demonstrate that our approach is valuable by finding join optimization bugs overlooked by existing approaches.

DBMSs versions. We tested the latest development version of each DBMS. When the code of the nightly version of a DBMS was updated, we rebuilt the DBMS from its official repository and then tested it. Specifically, we started testing MySQL from commit `1c7039a8`, SQLite from check-in (commit) `394bef5441` and later commits, TiDB from commit `21f9bb7` and later commits, DuckDB from commit `c87ae7a` and later commits, MonetDB from commit `d8999a3` and later commits, PostgreSQL from commit `a3ef0b57`, ClickHouse from commit `0d6c79e4`, CrateDB from commit `9879c7e` and later commits, Dolt from commit `ab5d58b` and later commits, and CockroachDB from version v25.2.0. Because Umbra's source code is unavailable, we tested its Docker version starting from 25.05 `sha256:8a2daea5`, which was the latest version at the time of testing.

5.2 Bug Finding Effectiveness

First, to answer Q1, we conducted a testing campaign aiming to find overlooked bugs in DBMSs.

Testing methodology. We conducted large-scale testing on 11 DBMS for more than three months, followed by a few additional months of intermittent testing during the implementation of our approach and the writing of the paper. Such an evaluation methodology is common and was also used in other research works [20, 46, 49]. Before reporting bugs, we manually reduced each

[8] https://db-engines.com/en/ranking

bug-inducing test case to a minimal bug-triggering query [45]. The reduction iteratively removed statements or features, while preserving bug reproducibility until a minimal case was obtained. The reduction time depended on test-case complexity, with an average of 30 minutes per case. This process could be automated using existing test-case reduction tools [27]. Our goal was to balance uniqueness and timeliness in bug reporting: we aimed to quickly report bugs that were likely unknown to facilitate timely fixes, while also avoiding reporting duplicate bugs that could burden developers. Specifically, we reported crash bugs with previously unseen stack traces or error messages, even when some crash bugs we reported had not been fixed. When we found a suspected new join logic bug-inducing case, we analyzed its reduced version for new features, compared to the past unfixed logic bugs we reported. When we found a new feature to be necessary to trigger the bug, such as an index, a view, or a different join type, we reported the bug-inducing test case. Due to our in-depth analysis, only 2 bugs of DuckDB and 1 bug of SQLite we reported were marked as duplicates by the developers. For some DBMSs (*e.g.*, MySQL), we stopped reporting bugs, because we observed that ours and previously reported bugs remained unfixed, which leads to many long-accumulated stale bugs, which was also observed in previous works [46, 49].

Bug statistics. As illustrated in Table 1, we have successfully found 103 join bugs in 11 DBMSs, with 100 being unique, previously unknown join bugs, and 91 of these bugs have been fixed. Of the 103 found bugs, 69 were join logic bugs and 34 were join internal errors or crashes. SQLite is typically seen as one of the most challenging test targets, as it has already been thoroughly tested by various logic bug detection approaches such as PQS [37], NoREC [35], TLP [36], EET [20], and CODDTest [46], with all of the previously reported bugs being quickly fixed by developers.

Notably, our approach found 14 previously unknown join bugs in SQLite, all of which have been fixed, showing our approach's effectiveness. Although EET [20] improved join testing by finding 7 join logic bugs in SQLite, it failed to detect 13 join logic bugs of SQLite found by JIR. Additionally, because of the absence of query hint support in SQLite, DuckDB, MonetDB, and Umbra, DQP [5] cannot be effectively evaluated on these systems, as it utilizes query hints for testing. While TQS was not evaluated on SQLite, which is recognized as one of the most challenging testing targets, all of the 13 join logic bugs we found in SQLite belong to non-equi-joins, which are outside the scope of TQS. In contrast, our method, JIR, is independent of query hints and capable of detecting join bugs beyond equi-joins, thereby successfully detecting join logic bugs in all these systems. This demonstrates JIR is effective for detecting join bugs across DBMSs as long as they support joins.

Bug significance. Bugs that trigger active developer concern or widespread discussions can often reflect the importance of bugs. Specifically, how developers perceive and respond to it provides valuable insight into the impact of bugs. We collected developers' feedback from MySQL, SQLite, DuckDB, and TiDB; MySQL and TiDB have clear classifications for bug severity levels, while the authors of DuckDB and SQLite were highly responsive in addressing our bug reports. Among the 2 logic bugs found by JIR in MySQL, 1 was recognized as a serious bug, and another was perceived as a critical bug. Of the 5 bugs fixed by TiDB, two were flagged as critical, and two were labeled as major; more impressively, 1 join crash bug stirred heated discussions that resulted in 25 code file changes related to the TiDB optimizer and nearly 40 commits to fix this critical bug.[9]

We also received positive feedback from the DBMS developers. For example, DuckDB and SQLite developers shared their appreciation and acknowledgments.

> ***DuckDB***: Awesome! I was wondering if this query was automatically generated somehow!

[9]https://github.com/pingcap/tidb/issues/60625

Table 1. Our approach helps us to find 100 unique, previously unknown join bugs in 11 DBMSs, of which 91 were fixed by developers, and of which 69 were join logic bugs.

DBMS	Bug Type		Status		
	Join Logic Bug	Join Internal Error/Crash	Fixed	Verified	Duplicated
SQLite	13	1	13	0	1
MySQL	2	0	0	2	0
CockroachDB	0	1	0	1	0
ClickHouse	0	1	1	0	0
DuckDB	11	10	16	3	2
TiDB	6	2	5	3	0
MonetDB	17	13	30	0	0
Umbra	2	6	8	0	0
Dolt	15	0	15	0	0
CrateDB	3	0	3	0	0
PostgreSQL	0	0	0	0	0
Total	69	34	91	9	3

SQLite: This is a problem in the new `EXISTS-to-JOIN` optimization which has not appeared in any official release. It is always best to catch these kinds of problems before a release rather than afterwards. So thanks for that!

False alarms. While evaluating our approach on SQLite, we encountered a SQLite quirk,[10] which resulted in false alarms of JIR. Specifically, SQLite allows non-aggregated columns to appear in the `SELECT` clause of aggregate queries, while such queries will be rejected by other systems. When such an aggregate query serves as a subquery to form a join in conjunction with using non-aggregated columns to construct join conditions, such columns may introduce non-deterministic results. Such ambiguous queries have also been noted, but not systematically handled by other works [5, 35, 36]. After identifying this issue, we believe that if we manually adjusted our query generator to avoid generating non-aggregated columns in aggregate queries, this false alarm can be effectively eliminated.

5.3 Diversity of Join Optimization Bugs

We answered Q2 through an in-depth analysis of the 39 join optimization bugs discovered in SQLite, DuckDB, TiDB, and MySQL, examining the diversity of join-related features in the queries that triggered these bugs.

Join optimization features. We summarize the features of the selected join bugs in Table 2. In detail, the column *Subquery* shows that 20 bugs required joins involving subqueries, and notably, 8 bugs are triggered by `LATERAL JOIN`, in which RHSs of joins are subqueries referring to columns from LHSs. The column *Join type* illustrates the join types for triggering bugs; notably, 18 bugs contain multiple join types, indicating that the corresponding necessary bug-triggering cases involve complex join queries that consist of more than 1 join type. In addition, the column *Index usage* means that an incorrect index usage is necessary to induce optimization bugs. Besides, 2 buggy queries of SQLite invoke window functions (*e.g.*, `TOTAL()`) in conjunction with joins to

[10] See https://www.sqlite.org/quirks.html, Item 6: *Aggregate Queries Can Contain Non-Aggregate Result Columns That Are Not In The GROUP BY Clause.*

trigger the bugs. Overall, 37 of the 39 buggy joins contain non-equi-join conditions, and only 2 were equi-join logic bugs.

Table 2. Join and optimization features of the queries triggering logic or crash bugs

Bug ID	DBMS	Bug type	Subquery	Join type	Index usage	Join Condition
fe8775443d	SQLite	Logic	●	INNER JOIN	❍	Non-Equal
35bf50ea01	SQLite	Logic	●	INNER JOIN LEFT JOIN	❍	Non-Equal
a8704b30f3	SQLite	Logic	❍	INNER JOIN RIGHT JOIN	●	Non-Equal
7dee41d325	SQLite	Logic	❍	RIGHT JOIN LEFT JOIN	●	Non-Equal
4fc70203b6	SQLite	Logic	●	RIGHT JOIN NATURAL LEFT JOIN	❍	Non-Equal
5028c785b6	SQLite	Logic	❍	FULL JOIN NATURAL JOIN NATURAL LEFT JOIN	❍	Non-Equal
829306db47	SQLite	Logic	●	RIGHT JOIN NATURAL JOIN NATURAL LEFT JOIN	❍	Non-Equal
68f29a2005	SQLite	Logic	●	INNER JOIN NATURAL FULL JOIN NATURAL LEFT JOIN NATURAL CROSS JOIN	❍	Non-Equal
3e45ed31d8	SQLite	Logic	❍	ANTI JOIN	❍	Non-Equal
428ef7c468	SQLite	Logic	❍	LEFT JOIN	❍	Non-Equal
6e1d387c7e	SQLite	Logic	❍	SEMI JOIN	❍	Non-Equal
ccfb3b5052	SQLite	Internal Error	●	NATURAL RIGHT JOIN INNER JOIN CROSS JOIN	❍	Non-Equal
16783	DuckDB	Crash	❍	ANTI JOIN	❍	Non-Equal
17335	DuckDB	Crash	●	CROSS JOIN RIGHT JOIN LATERAL JOIN	❍	Non-Equal
17446	DuckDB	Crash	●	CROSS JOIN LEFT JOIN RIGHT JOIN LATERAL JOIN	❍	Non-Equal
17700 18049	DuckDB	Crash	●	CROSS JOIN INNER JOIN LATERAL JOIN	❍	Non-Equal
17701 18000	DuckDB	Crash	●	CROSS JOIN LATERAL JOIN	❍	Non-Equal
17440	DuckDB	Internal Error	❍	INNER JOIN SEMI JOIN	❍	Non-Equal
17378	DuckDB	Internal Error	●	INNER JOIN	❍	Non-Equal
16803	DuckDB	Logic	❍	ANTI JOIN	❍	Non-Equal
16901	DuckDB	Logic	●	RIGHT JOIN	❍	Equal
17217	DuckDB	Logic	❍	INNER JOIN	❍	Non-Equal
17372	DuckDB	Logic	●	INNER JOIN RIGHT JOIN LATERAL JOIN	❍	Non-Equal
17380	DuckDB	Logic	❍	INNER JOIN	●	Non-Equal
17561	DuckDB	Logic	❍	INNER JOIN	❍	Non-Equal
17792	DuckDB	Logic	●	LEFT JOIN CROSS JOIN LATERAL JOIN	❍	Non-Equal
17417	DuckDB	Logic	❍	ANTI JOIN	❍	Non-Equal
60322	TiDB	Logic	●	LEFT JOIN INNER JOIN	❍	Non-Equal
60625	TiDB	Crash	●	INNER JOIN	❍	Non-Equal
60489	TiDB	Internal Error	●	INNER JOIN	❍	Non-Equal
60659	TiDB	Logic	●	RIGHT JOIN INNER JOIN	❍	Non-Equal
61306	TiDB	Logic	❍	INNER JOIN	●	Non-Equal
61327	TiDB	Logic	●	LEFT JOIN INNER JOIN	❍	Non-Equal
60093	TiDB	Logic	●	INNER JOIN	❍	Non-Equal
60217	TiDB	Logic	❍	INNER JOIN	❍	Non-Equal
118014	MySQL	Logic	❍	LEFT JOIN	❍	Equal
118063	MySQL	Logic	❍	ANTI JOIN	●	Non-Equal

We investigated join types appearing in these bug-triggering queries, and encouragingly, our found join logic bugs span **`INNER JOIN`**, **`LEFT JOIN`**, **`RIGHT JOIN`**, **`CROSS JOIN`**, **`FULL JOIN`**,

Listing 2. TiDB Bug #60322: An incorrect join reordering for a multiple join with a subquery

```
1  CREATE TABLE t0(c0 BOOL);
2  CREATE TABLE t1(c0 CHAR);
3  INSERT INTO t1 VALUES (NULL);
4  INSERT INTO t0 VALUES (FALSE);
5  SELECT * FROM t1 LEFT JOIN (SELECT 0 AS col_0 FROM t0) as subQuery1 ON
       (subQuery1.col_0 = t1.c0) LEFT JOIN t0 ON (subQuery1.col_0 <=> t0.c0);
6  -- {(NULL, NULL, NULL)} ✔
7  SELECT * FROM t1 LEFT JOIN (SELECT 0 AS col_0 FROM t0) as subQuery1 ON
       (subQuery1.col_0 = t1.c0) RIGHT JOIN t0 ON (subQuery1.col_0 <=> t0.c0);
8  -- {(NULL, NULL, 0)} ✔
9  SELECT * FROM t1 LEFT JOIN (SELECT 0 AS col_0 FROM t0) as subQuery1 ON
       (subQuery1.col_0 = t1.c0) INNER JOIN t0 ON (subQuery1.col_0 <=> t0.c0);
10 -- {(NULL, NULL, NULL)} 🐞 -- Empty Result ✔
```

`NATURAL JOIN`, `SEMI JOIN`, `ANTI JOIN` and `LATERAL JOIN`, showing that our approach can detect join logic bugs triggered by a wide range of join types, as long as they are supported by the DBMS being tested. Moreover, these diverse join types, along with other diverse features, including subqueries, indexes, window functions, and non-equi-join predicates, demonstrate the generality of JIR, which can be effectively scaled to joinable LHSs and RHSs containing various features. In contrast, TQS [44] and DQP [5] have limited scalability when facing diverse join features. Specifically, developers implementing TQS need to carefully implement the join ground-truth simulator and handle semantic variations of these features across DBMSs. The implementation of DQP requires developers to thoroughly understand the synergy among query patterns, divergent query hint usage, and optimizer behaviors in different DBMSs.

5.4 Representative Bugs and Root Causes

Next, we present a selection of join logic bugs discovered by JIR, along with a root cause analysis.

TiDB bug on incorrect join reordering. Listing 2 shows an logic bug of a multiple-join query ending with an `INNER JOIN`. Suppose that the DBMS returns the correct result for the first and second query, ending with an `LEFT JOIN` and an `RIGHT JOIN`. We can infer that if we replace the last join as an `INNER JOIN` but with the same join condition, the expected result should be empty, because the result of an `INNER JOIN` should appear in both the results of an `LEFT JOIN` and an `RIGHT JOIN`, with shared join conditions. However, when TiDB evaluated the third query, it wrongly performed a join reordering rule concerning the `NULL`-safe operator in join conditions, resulting in an incorrect result. The TiDB developers labeled this bug as critical and quickly fixed this optimization rule.

SQLite bug on incorrect partial index usage in inner join. Listing 3 shows a join logic bug, containing a complex, but necessary join chain to trigger this bug. The second query ends with a `ANTI JOIN` and returns an empty result. The third query ends with an `INNER JOIN`, resulting in an empty set. Assuming that the execution results of the `INNER JOIN` and `ANTI JOIN` are both correct, JIR can infer that the expected result of the `LEFT JOIN` should also be empty, which yet contradicted with SQLite's execution result, indicating a join logic bug. The root cause of this bug was that SQLite's optimizer incorrectly used the partial index `P` created on the `t1` when evaluating `INNER JOIN`, although the `WHERE` clause is always `FALSE` in the partial index `P`, which actually makes `P` an empty index. Finally, the `INNER JOIN` leveraging this index will produce an empty result, because an empty

Listing 3. An incorrect index usage in an INNER JOIN

```
CREATE TABLE t0(c0 INT);
CREATE TABLE t1(c0 INT);
CREATE TABLE t2(c0 INT);
INSERT INTO t1 VALUES (1);
CREATE UNIQUE INDEX P ON t1((c0 NOT BETWEEN (c0 AND c0) AND c0)) WHERE c0 IN ();
SELECT * FROM t2 INNER JOIN t0 ON (t2.c0 IN ()) FULL JOIN (SELECT 0) as subQuery1
    ON TRUE LEFT JOIN t1 ON TRUE;
-- {(NULL, NULL, 0, 1)} ✔
SELECT * FROM t2 INNER JOIN t0 ON (t2.c0 IN ()) FULL JOIN (SELECT 0) as subQuery1
    ON TRUE WHERE NOT EXISTS (SELECT 1 FROM t1 WHERE TRUE); -- Empty Result ✔
SELECT * FROM t2 INNER JOIN t0 ON (t2.c0 IN ()) FULL JOIN (SELECT 0) as subQuery1
    ON TRUE INNER JOIN t1 ON TRUE;
-- Empty Result 🐞 -- {(NULL, NULL, 0, 1)} ✔
```

Listing 4. An incorrect index usage in an RIGHT JOIN

```
CREATE TABLE t1(c0 INT);
CREATE TABLE t2(c0 BOOLEAN);
CREATE TABLE t3(c0 INT);
INSERT INTO t1(c0) VALUES(1);
INSERT INTO t2(c0) VALUES(FALSE);
INSERT INTO t3(c0) VALUES(3);
CREATE INDEX t2i ON t2(c0) WHERE c0;
SELECT t1.c0, t2.c0 FROM t1 RIGHT JOIN t2 ON t2.c0 WHERE NOT EXISTS (SELECT 1
        FROM t3 WHERE t2.c0); -- {(NULL, 0)} ✔
SELECT t1.c0, t2.c0, t3.c0 FROM t1 RIGHT JOIN t2 ON t2.c0 LEFT JOIN t3 ON t2.c0;
-- {(NULL, 0, 3)} 🐞 -- {(NULL, 0, NULL)} ✔
```

index provides no rows to join. The SQLite developers quickly fixed this bug by disabling the use of a partial index when its `WHERE` clause does not depend on any columns of the indexed table.

SQLite bug on incorrect partial index usage in right join. Listing 4 shows a complex join logic bug due to incorrect partial index usage in a `RIGHT JOIN`. According to the result of `ANTI JOIN` and assuming its correctness, JIR can infer that the ground-truth result of the second query should contain that of `ANTI JOIN` concatenating with `NULL` as the second query changes the last join as `LEFT JOIN` yet keeps the same join condition. The culprit of this bug was incorrect partial index `t2i` usage of the table `t2`, and such misuse was triggered by the necessary join `LEFT JOIN t3 on t2.c0`. More significantly, this bug found by JIR extends the reach of a bug found by NoREC nearly 6 years ago.[11] However, NoREC was proposed in 2020 and has been used as an oracle in SQLRight [26], QPG [4], and SQLancer++ [49], all of which were used to test SQLite, but failed to find this extension bug. Richard Hipp, the creator of SQLite, quickly fixed this long-hidden bug by adding a simple condition to an `if` branch of the relevant code fragment, thereby disabling the use of the partial index on table `t2` when it served as the right-hand side of not only a `LEFT JOIN`, but also a `RIGHT JOIN`.

SQLite bug in both semi join and anti join with r-tree virtual table. Listing 5 shows a relatively simple test case that triggered a bug in both `SEMI JOIN` and `ANTI JOIN`. Given the execution results of `INNER JOIN` and `ANTI JOIN`, JIR predicts the oracle of `LEFT JOIN` should be

[11] https://sqlite.org/src/info/623eff57e76d45f6

Listing 5. A bug both appears in SEMI and ANTI JOIN

```
CREATE TABLE t0(c0);
CREATE VIRTUAL TABLE t1 USING rtree(c0, c1, c2, +c3 BLOB);
INSERT INTO t0(c0) VALUES (NULL);
INSERT INTO t0(c0) VALUES (1);
INSERT INTO t1(c0, c2, c3) VALUES (1, 1, 2);
SELECT t0.c0 FROM t0 LEFT JOIN t1 ON (t1.c3)OR(t0.c0 ISNULL);
-- {(NULL), (1)} ✔
SELECT t0.c0 FROM t0 INNER JOIN t1 ON (t1.c3)OR(t0.c0 ISNULL);
-- {(NULL), (1)} ✔
SELECT t0.c0 FROM t0 WHERE NOT EXISTS (SELECT 1 FROM t1 WHERE (t1.c3)OR(t0.c0
    ISNULL));
-- {(1)} 🐞 -- Empty Result ✔
SELECT t0.c0 FROM t0 WHERE EXISTS (SELECT 1 FROM t1 WHERE
        (t1.c3)OR(t0.c0 ISNULL));-- {(NULL)} 🐞 -- {(NULL), (1)} ✔
SELECT t0.c0 FROM t0 WHERE (EXISTS (SELECT 1 FROM t1 WHERE ((t1.c3)OR(t0.c0
    ISNULL))) ISNULL);
-- Empty Result ✔
```

$\{(NULL), (1), (1)\}$, but the actual execution result shows a conflict. After analysis, this bug appeared both in **ANTI JOIN** and **SEMI JOIN**. The SQLite developers promptly fixed it by invoking a state-reset function `sqlite3_reset` in the `rtree` implementation.[12] More importantly, when we used TLP-style transformation to reproduce this bug as shown in the last three queries, the combination result of their TLP variants conforms to that of the unpartitioned query **SELECT** `t0.c0` **FROM** `t0`. This bug indicates that TLP fails to detect bugs under certain predicates on which some variant queries each yield buggy results, but such incorrect results neutralize mutually, causing the overall result to remain consistent with the test oracle's expectations.

DuckDB bug on incorrect optimization for join condition involving float columns. Listing 6 shows a multiple join logic bug in DuckDB due to its incorrect optimization for join conditions involving float columns. Leveraging the assuming-correct execution results of **INNER JOIN** and **ANTI JOIN**, JIR infers the ideal result of **LEFT JOIN** and detects this bug. The root cause was a lack of support for equi-joins involving float columns. To address this, the DuckDB developers introduced safeguards to disable optimizations to be performed for such join conditions.

DuckDB bug on incorrect statistics propagation for anti-join on empty tables. Listing 7 illustrates a query that triggers an **ANTI JOIN** join logic bug in the underlying statistics of data storage. Ideally, derived from the execution results of **LEFT JOIN** and **INNER JOIN**, the result of **ANTI JOIN** should conform to that **LEFT JOIN**. However, when handling **ANTI JOIN**, DuckDB's statistics module failed to make accurate estimations for empty tables, and thus returned incorrect results for **ANTI JOIN** queries when the right-hand side was empty. The DuckDB developers fixed this bug by adding a utility function in the statistics module to determine whether the statistics match an empty table.

DuckDB and Umbra bug on incorrect correlated subquery unnesting in lateral join. Listing 8 show a logic bug in **LATERAL JOIN** we found. As shown in the first query, using an **INNER JOIN LATERAL** between t0 and a subquery that references both t0 and t2 returned an empty result. Then, when we expressed the lateral version **ANTI JOIN** via **NOT EXISTS** (as shown in the second query), it returned the row (2, 1). However, the third query, which uses **LEFT JOIN LATERAL**, exposes a contradiction

[12]https://sqlite.org/src/info/3c0afda372

Listing 6. DuckDB Bug#16901: An multi-join logic bug due to wrong optimization for joining on float columns

```
CREATE TABLE t0(c0 DOUBLE);
CREATE TABLE t1(c0 DOUBLE);
CREATE TABLE t2(c0 DOUBLE);
INSERT INTO t0(c0) VALUES (0.0);
INSERT INTO t1(c0) VALUES ('-0.0');
INSERT INTO t2(c0) VALUES (0.0);
SELECT sub.s1, t2.c0 FROM t2 INNER JOIN
(
  SELECT t0.c0 as s1, t1.c0 as s2
  FROM t0 RIGHT JOIN t1 ON t0.c0 = t1.c0 WHERE NOT t0.c0
) as sub ON CAST(sub.s1 AS TEXT) = CAST(t2.c0 AS TEXT);
-- {(0.0, 0.0)} ✔
SELECT t2.c0 FROM t2 WHERE NOT EXISTS
(
    SELECT 1 FROM (SELECT t0.c0 as s1, t1.c0 as s2 FROM t0 RIGHT JOIN t1
      ON t0.c0 = t1.c0 WHERE NOT t0.c0) as sub
        WHERE CAST(sub.s1 AS TEXT) = CAST(t2.c0 AS TEXT)
); -- Empty Result ✔
SELECT sub.s1, t2.c0 FROM t2 LEFT JOIN
(
  SELECT t0.c0 as s1, t1.c0 as s2
  FROM t0 RIGHT JOIN t1 ON t0.c0 = t1.c0 WHERE NOT t0.c0
) as sub ON CAST(sub.s1 AS TEXT) = CAST(t2.c0 AS TEXT);
-- {(NULL, 0.0)} 🐞 -- {(0.0, 0.0)} ✔
```

Listing 7. DuckDB Bug #17417: An `ANTI JOIN` logic bug due to wrong statistic information

```
CREATE TABLE t0(c0 INT);
CREATE TABLE t2(c0 INT);
INSERT INTO t0(c0) VALUES (5);
SELECT * FROM t0 LEFT JOIN t2 ON (t2.c0 >= t0.c0);
-- {(5, NULL)} ✔
SELECT * FROM t0 INNER JOIN t2 ON (t2.c0 >= t0.c0);
-- Empty Result ✔
SELECT * FROM t0 ANTI JOIN t2 ON (t2.c0 >= t0.c0);
-- Empty Result 🐞 -- {(5, NULL)} ✔
```

with the combination of **`INNER JOIN`** and **`ANTI JOIN`**. The simplified bug-triggering test case contains a **`CROSS JOIN`** between t2 and t0, as well as a **`LEFT LATERAL JOIN`** with a subquery, in which 2 lateral references t0.c0 and t2.c0 are retrieved from the appeared table t0 and t2, and this logic bug cannot be triggered if we remove any table or lateral reference. This bug was so elusive that it was caused by just one line of code related to DuckDB optimizer's correlated subquery unnesting, especially in dependent join flattening. Specifically, when a dependent join node had a child that was also a dependent join, the DuckDB optimizer incorrectly computed the offset which was necessary to bind the correlated columns from the outer query to their corresponding positions in the subquery. Additionally, this underlying cause also led to other two crash bugs of DuckDB related to lateral join execution. Notably, the same test case also exposed a logic bug in Umbra.

Listing 8. DuckDB Bug#17792 and Umbra Bug: An LATERAL JOIN logic bug due to incorrect subquery unnesting

```
CREATE TABLE t0(c0 INT);
CREATE TABLE t2(c0 INT);
INSERT INTO t0(c0) VALUES (1);
INSERT INTO t2(c0) VALUES (2);
SELECT * FROM t2, t0 INNER JOIN LATERAL(SELECT t0.c0 AS col_0,t2.c0 AS col_1) as
    subQuery1 ON (subQuery1.col_1 < t0.c0); -- Empty Result ✔
SELECT * FROM t2, t0 WHERE NOT EXISTS(SELECT 1 FROM (SELECT t0.c0 AS col_0,t2.c0
    AS col_1) as subQuery1 WHERE (subQuery1.col_1 < t0.c0)); -- {(2, 1)} ✔
SELECT * FROM t2, t0 LEFT JOIN LATERAL(SELECT t0.c0 AS col_0, t2.c0 AS col_1) as
    subQuery1 ON (subQuery1.col_1 < t0.c0);
-- DuckDB Buggy Result: Empty Result 🐞
-- Umbra Buggy Result: {(2, 1, 1, NULL)} 🐞
-- {(2, 1, NULL, NULL)} ✔
```

5.5 Comparative Study of Test Oracles

In this section, we chose EET [20] and DQP [5] for experimental comparison. Although we have demonstrated that JIR can find join logic bugs on systems lacking support for query hints, we reimplemented DQP for Dolt for a deeper comparison. We omitted TQS, because it is neither publicly available, nor can it be applied to non-equi-join logic bugs; we have shown 25 such bugs in Table 2.

Comparison with EET. We conducted this bug-finding comparison in SQLite check-in 394bef5441, on which we began our evaluation of our approach. It is a reasonable choice, as we found latent join bugs whose inducing commits date back to 2017, 2020, 2022, and 2023, as shown in Table 3, all of which precede the publication of EET. If EET fails to detect these bugs, it suggests that these bugs are missed by EET. For JIR, we set 5 testing threads in SQLancer for a duration of 12 hours. For EET, we set 5 Docker instances for testing according to the tool's specification, and also ran them for 12 hours. This is because we wanted to make a fair comparison with EET, and each EET testing instance may cost about 10 GB of memory,[13] so we set the testing threads or instances to 5. SQLancer provides a MaxDepth option to control the maximum depth of an expression, and we used the default configuration of SQLancer, in which this option is set to 3. During the 12-hour test, EET and JIR detected 0 and 5 unique join logic bug respectively. Although JIR failed to detect all reported logic bugs within 12 hours, we believe it is reasonable. Firstly, generating certain corner cases can be time-consuming, so a 12-hour test may be insufficient to produce queries that trigger previously reported bugs. Secondly, two bugs of Table 2 were introduced in July 2025, making them undetectable in earlier versions. We also conducted an empirical analysis for the earliest bug-inducing commits of SQLite's trunk branch; such a method was also employed in EET [20] and CODDTest [46]. We selected the 10 join logic bugs, 35bf50ea01 to 6e1d387c7e, from Table 2 for analysis. This is because fe8775443d and 35bf50ea01 share the same bug-fixing commit, and both JIR and EET are oracles for logic bug detection. The results are summarized in Table 3. Among these bugs, only 3e45ed31d8 and 6e1d387c7e were introduced in July 2025, while the other 8 bugs were introduced before 2024, the year EET was published, with the longest existing for 8.5 years. Notably, JIR discovered three latent join bugs within the same commit dated 2022-04-21, in which SQLite introduced `RIGHT JOIN` and `FULL JOIN`. In contrast, EET failed to detect these bugs. This

[13] See https://github.com/JZuming/EET/blob/main/docs/test/sqlite_test_latest.md.

is because JIR directly targets join semantics and shifts the granularity of predicate evaluation across a wide range of join types, enabling it to successfully find multiple join logic bugs missed by existing approaches, even latent within a single commit for more than 3 years. Therefore, the 12-hour test results and bug-inducing commit time analysis demonstrate that JIR can detect latent join logic bugs that are missed by EET.

Table 3. Latency of the SQLite join logic bugs found by JIR

Bug ID	Inducing Commit Time	Latency (Years)
35bf50ea01	2020-01-16	5.7
a8704b30f3	2022-04-21	3.5
7dee41d325	2023-11-10	1.9
4fc70203b6	2022-04-21	3.5
5028c785b6	2022-04-21	3.5
829306db47	2022-04-21	3.5
68f29a2005	2022-05-25	3.4
3e45ed31d8	2025-07-01	0.3
428ef7c468	2017-04-13	8.5
6e1d387c7e	2025-07-08	0.3

Comparison with DQP. To empirically compare with DQP, we implemented both DQP and JIR on top of SQLancer++ [49] for testing Dolt, which, in terms of query hints, supports seven types of join hints and no other kinds of hints. The complete query hint set of Dolt makes it a particularly suitable testbed for a fair comparison between DQP and JIR. In particular, Dolt's complete query hint set consists of `JOIN_ORDER`, `LOOKUP_JOIN`, `MERGE_JOIN`, `HASH_JOIN`, `INNER_JOIN`, `SEMI_JOIN`, and `ANTI_JOIN`, all of which are exclusively related to join, eliminating confounding factors from non-join-related hints. Because these join hints cannot be applied in joins with subqueries, we disabled the generation of joins with subqueries in SQLancer++. We chose the commit `ab5d58b` of Dolt as the baseline for our bug-finding campaign. Starting from this commit, we reported 12 unique bugs found by JIR, among which 10 have been fixed, and 3 unique bugs found by DQP, all of which have been fixed. In addition, we conducted a structured, timeboxed experiment on that commit of Dolt, for a duration of 3 hours and a single testing thread in SQLancer++. We could not set a longer duration, as it caused Dolt to crash during execution. Figure 3 illustrates the unique bugs discovered by JIR and DQP in this experiment. Specifically, JIR discovered 7 unique bugs, while DQP discovered 4 unique bugs; among them, 2 were discovered by both oracles. Hence, JIR is complementary to DQP and can discover several join logic bugs that are missed by DQP.

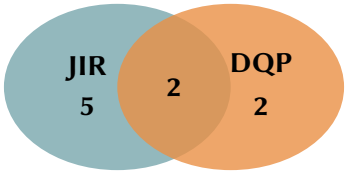

Fig. 3. The number of unique bugs detected by oracles.

6 Discussion

Extensibility. In this paper, we introduce six implication reasoning rules for detecting join logic bugs, demonstrating the effectiveness of JIR. Beyond these initial rules, we envision incorporating many additional ones to further enhance bug-finding capabilities, as long as such rules preserve the correctness of the join semantics. For example, more fine-grained implication rules can be devised for `INNER JOIN`. In particular, this equivalence can be leveraged to reason about the expected output of an `INNER JOIN` whose join condition is expressed as a conjunction of multiple predicates: `A INNER JOIN B ON P and Q = (A INNER JOIN B ON P) INTERSECT ALL (A INNER JOIN B ON Q)`. However, using `INTERSECT` or `INTERSECT ALL` introduces limitations. Firstly, `INTERSECT` performs deduplication, which requires additional `DISTINCT` operations and is thus only applicable to set semantics. Secondly, while `INTERSECT ALL` preserves duplicate rows, it is not universally supported across DBMSs. For instance, SQLite, MySQL, and TiDB lack support for `INTERSECT ALL`. Finally, both `INTERSECT` and `INTERSECT ALL` can only be applied to queries of the form `SELECT *`, making it unsuitable for validating queries that involve implicit row identifiers or explicit column projections, since identical values in explicit columns of distinct rows can lead to false alarms.

Compatibility with WHERE. The `WHERE` clause serves as a filter on the output of join queries, without affecting the function of the `JOIN ON` of different join types. Additionally, `SEMI JOIN` and `ANTI JOIN` are commonly expressed using predicates (`EXISTS` and `NOT EXISTS`) in the `WHERE` clause; thus, these predicates can be conjoined with other `WHERE` predicates by `AND`. Consequently, the reasoning rules presented in this paper remain applicable under arbitrary `WHERE` conditions. Specifically, we can firstly apply JIR to infer the oracle of the join result and then apply the `WHERE` filter. This property ensures that JIR can handle complex queries that combine multiple joins with `WHERE` predicates.

Although generating `WHERE` predicates in our evaluation, in which we found 103 join bugs, there were only 2 internal issues related to the interaction between `JOIN` and an additional `WHERE` predicate, indicating that `WHERE` predicates (except `EXISTS` and `NOT EXISTS`, which is necessary in `SEMI JOIN` and `ANTI JOIN`) are not necessary bug-inducing features of the other 101 bugs, and our approach focuses testing `ON` semantics of `JOIN`.

Long-term Implication. In the long term, we believe that our technique remains significant. Based on the analysis of bug reports from the OSSFuzz continuous fuzzing platform [39], prior research has shown that bugs are introduced at a constant rate and that approximately 80% of them are regression bugs caused by recent code changes [50]. This phenomenon also applies to DBMSs. For example, SQLite introduced a new optimization rule for evaluating `AND` and `OR` operators on October 29 2025, and JIR subsequently detected a new join logic bug in this commit, which was promptly fixed. Therefore, our approach remains long-term effective.

Testing Scope. Conceptually, JIR is grounded on relational join semantics, and it triggers join logic bugs by checking evaluation inconsistencies of the same join predicate across different relational join types. Formally, the basic form of a JIR testing query is `SELECT * FROM A JOIN B ON P`, where A and B represent any joinable datasources (*e.g.*, tables, views, and subqueries), and the join predicate `P` is preserved, but `JOIN` is varied among different relational join types. In comparison, existing test oracles, such as NoREC [35], TLP [36], Pinolo [15] leverage predicates on `WHERE` clause to trigger logic bugs; EET [20] leverages equivalent expression transformations on any SQL clause supporting expressions to trigger logic bugs. As a result, JIR's design choice on checking predicate semantic consistency across join types makes it a testing oracle that specializes in detecting join logic bugs. Also, it can be used to detect bugs in parallel or distributed join processing. Intuitively, when

two semantically related joins execute concurrently or in a distributed manner while producing semantic-conflicting results on the same database, it indicates the presence of parallel or distributed join logic bugs. However, JIR cannot test special join types that go beyond standard relational joins, such as `ASOF JOIN` in DuckDB.

7 Related Work

Logic Bug Detection for DBMSs. In recent years, a plethora of logical bug detection methods for DBMSs have been proposed. They can be classified into semantics-emulation approaches and metamorphic testing approaches.

Semantics-emulation approaches validate the correctness of query processing by reconstructing the expected outputs through a self-implemented oracle computation engine. Formally, given a query Q over a database instance D, the oracle engine O simulates or re-implements the semantics of each operator in Q to derive the expected result $R^* = O(Q, D)$. The actual output $R = DB(Q, D)$, where DB denotes the DBMS under test, is then compared against R^*. Any discrepancy between R and R^* indicates a potential logical bug in the DBMS implementation.

PQS [37] and TQS [44] belong to semantics-emulation approaches. PQS generates queries containing predicate expressions that are guaranteed to evaluate to `TRUE` for a selected pivot row, according to its self-implemented expression evaluator that conforms to the expression semantics of the tested DBMSs. If a tested DBMS fails to return the expected row, PQS flags this as a logic bug, indicating a deviation from the correct evaluation semantics of that expression. TQS simulates join semantics itself, thereby providing a join oracle, but it is limited only to PK-FK equi-join under set semantics. In practice, many joins are many-to-many and naturally produce duplicates, yet it is unclear how TQS can simulate duplication under bag semantics.

Semantic simulation requires re-implementing the semantics of specific operators or expressions, which often vary across DBMSs. Its advantage is that, assuming the simulator is perfectly implemented, it can serve as a universal oracle across different DBMSs. However, semantic simulation has a high implementation effort, and thus PQS is currently unmaintained.

Metamorphic testing [7] verifies the behavior of a program by generating new inputs derived from existing ones and their corresponding outputs, and finding bugs by inferring expected outputs of such new inputs via metamorphic relations. Specifically, given an input I and its output $O = P(I)$, where P is the program under test, a follow-up input I' is created such that the new output $O' = P(I')$ is expected to satisfy a known metamorphic relationship with the original output O.

NoREC [35], TLP [36], DQE [40], Pinolo [15], EET [20], and CODDTest [46] are all based on metamorphic-testing principles to detect logic bugs. NoREC [35] detects logic bugs by checking the semantic consistency of predicate evaluation between the `WHERE` and `SELECT` clauses. TLP [36] validates the three-valued logical completeness of predicate expression handling in SQL by decomposing an original query into three mutually exclusive ones: when `P`, when `NOT P`, and when `P IS NULL`, and assuming the union of these three queries should match the result of the original unpartitioned query. DQE [40] extends the evaluation of predicate semantics beyond the `SELECT` and `WHERE` clause by applying identical predicates across other SQL clauses such as `UPDATE` and `DELETE`. Discrepancies in the affected rows are determined as logic bugs. Pinolo [15] alters the query predicates to make them more permissive or restrictive, expecting the results to be supersets or subsets of the originals, thereby detecting logical bugs. EET [20], focusing on semantic preservation of expressions, iterates and transforms all predicate expressions of a query into semantically equivalent ones to construct new queries and detect logical bugs. CODDTest [46] assumes the semantic equivalence of queries after applying constant folding and propagation for expressions.

The core advantage of metamorphic testing is that it detects bugs by leveraging metamorphic relations, thus eliminating the need to implement a complex oracle simulator. However, metamorphic testing also has limitations. Firstly, its effectiveness depends on the soundness of the adopted metamorphic relations. Secondly, it will miss bugs if both the original and mutated inputs produce buggy outputs that still satisfy the metamorphic relation. In this context, semantic-simulation approaches can detect such bugs.

All above-mentioned metamorphic approaches have limitations in finding join logic bugs, as they do not target testing `JOIN ON` semantics, but rely on expression-level manipulations and transformations within other SQL clauses. Conversely, JIR is a general oracle specially designed for detecting join logic bugs, which directly checks the consistency of `JOIN ON` semantics across a wide range of join types and can work on any join-enabled relational DBMSs.

Test Case Generation for DBMSs. In addition to logical bug detection, several approaches [3, 12, 18, 24, 47] aim to generate test inputs (*i.e.*, sequences of SQL statements) that comprehensively stress the DBMS. SQLsmith [3], a representative and widely-used DBMS fuzzer, uses pre-defined AST rules of SQL and randomly generates queries to find crash bugs. SQUIRREL [47] is based on a new and expressive intermediate representation for SQL, on which it performs syntax-preserving mutation and semantics-guided instantiation, to find memory bugs of DBMSs. Griffin [12], a grammar-free DBMS fuzzing tool, summarizes DBMS state information metadata graphs to guide SQL mutations and thus improve semantic correctness. DynSQL [18] incorporates interactions into query generation, dynamically capturing the latest state information for complex and valid query generation. LEGO [24] proposes a type-affinity-driven SQL fuzzer that diversifies the types of SQL statements to improve the coverage of fuzzing. Additionally, some works [4, 26, 48] strengthen test-case generation yet retain a focus on logic bug detection, leveraging oracles such as NoREC [35] and TLP [36]. SQLRight [26] enhances logic bug detection by incorporating coverage-based guidance, enabling exploration of deeper execution paths. QPG [4] records query plans and mutates the database states toward previously unseen query plans, while maintaining general oracles to detect logic bugs. ShQveL [48] uses LLMs in its query generation process.

Transaction Bug Detection for DBMSs. Various approaches have been proposed to detect transactional bugs [11, 19, 21, 43]. Specifically, Troc [11] decomposes a transaction into independent statements and executes them on database views constructed according to the DBMSs' claimed isolation levels. TxCheck [19] decouples transactions into independent and dependent statements, then reschedules independent ones to generate semantically equivalent variants while keeping the order of dependent statements, and executes both the original and variant transactions to detect inconsistencies. Through transaction execution history, Elle [21] identifies isolation anomalies based on the Adya formalism [1], and Cobra [43] infers serializability violations in DBMSs. All the above-mentioned works have limited support for join generation inside transactions; for example, Cobra lacks support for the join operator. Our work is complementary to theirs and can be, conceptually, used to detect join logic bugs in the transaction processing context. For example, two read-only transactions execute semantically related joins, but observe semantic-violating results, indicating join bugs.

8 Conclusion

In this paper, we have proposed a novel, general, and highly effective method, Join Implication Reasoning (JIR), for finding join logic bugs in DBMSs. JIR does not rely on simulating join oracle or query hints, enabling the testing of join semantics for all common join types, under arbitrary join conditions, on join-supported systems. We evaluated JIR on 11 extensively tested DBMSs and found 100 unique, previously unknown join bugs, of which 91 were fixed by developers, and of

which 69 were join logic bugs. We showed that JIR can detect join logic bugs that are missed by existing methods. Given that JIR is a simple-to-realize, general, and effective approach, we envision JIR to be a widely adopted methodology for join logic bug detection.

Acknowledgments

This research is supported by the Ministry of Education, Singapore, under the Academic Research Fund Tier 1 (FY2023), the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing), and an Amazon Research Award Fall 2023. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the the Ministry of Education, Singapore, National Research Foundation, Singapore, Cyber Security Agency of Singapore, and Amazon.

Artifact Availability

The artifact of this paper is provided at https://doi.org/10.5281/zenodo.19000645.

References

[1] Atul Adya. 1999. Weak consistency: a generalized theory and optimistic implementations for distributed transactions. (1999).

[2] Foto N Afrati, Matthew Damigos, and Manolis Gergatsoulis. 2010. Query containment under bag and bag-set semantics. *Inform. Process. Lett.* 110, 10 (2010), 360–369.

[3] Andreas Seltenreich, Bo Tang, Sjoerd Mullender. 2018. SQLsmith: A random SQL query generator. https://github.com/anse1/sqlsmith.

[4] Jinsheng Ba and Manuel Rigger. 2023. Testing database engines via query plan guidance. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2060–2071.

[5] Jinsheng Ba and Manuel Rigger. 2024. Keep It Simple: Testing Databases via Differential Query Plans. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[6] 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.

[7] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. 1998. Metamorphic Testing: A New Approach for Generating Next Test Cases. *Technical Report* Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong (1998).

[8] Jim Diederich and Jack Milton. 1988. New methods and fast algorithms for database normalization. *ACM Transactions on Database Systems (TODS)* 13, 3 (1988), 339–365.

[9] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: A decision support benchmark for workload-driven and traditional database systems. *Proceedings of the VLDB Endowment* 14, 13 (2021), 3376–3388.

[10] James Dong and Fredrik Kjolstad. 2025. A Compiler for Operations on Relations with Bag Semantics. *arXiv preprint arXiv:2502.06988* (2025).

[11] Wensheng Dou, Ziyu Cui, Qianwang Dai, Jiansen Song, Dong Wang, Yu Gao, Wei Wang, Jun Wei, Lei Chen, Hanmo Wang, et al. 2023. Detecting isolation bugs via transaction oracle construction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1123–1135.

[12] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-free DBMS fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[13] César A Galindo-Legaria, Arjan Pellenkoft, and Martin L Kersten. 1995. Uniformly-distributed random generation of join orders. In *International Conference on Database Theory*. Springer, 280–293.

[14] Ahmad Ghazal, Alain Crolotte, and Ramesh Bhashyam. 2004. Outer join elimination in the Teradata RDBMS. In *International Conference on Database and Expert Systems Applications*. Springer, 730–740.

[15] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. 2023. Pinolo: Detecting logical bugs in database management systems with approximate query synthesis. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 345–358.

[16] Gerhard Hill and Andrew Ross. 2009. Reducing outer joins. *The VLDB Journal* 18, 3 (2009), 599–610.

[17] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[18] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. {DynSQL}: Stateful Fuzzing for Database Management Systems with Complex and Valid {SQL} Query Generation. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4949–4965.

[19] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting transactional bugs in database engines via {graph-based} oracle construction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 397–417.

[20] Zu-Ming Jiang and Zhendong Su. 2024. Detecting logic bugs in database engines via equivalent expression transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 821–835.

[21] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring isolation anomalies from experimental observations. *arXiv preprint arXiv:2003.10554* (2020).

[22] George Konstantinidis and Fabio Mogavero. 2025. Bag Containment of Join-On-Free Queries. In *28th International Conference on Database Theory (ICDT 2025)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 5–1.

[23] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[24] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. 2023. Sequence-oriented DBMS fuzzing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 668–681.

[25] Yu Liang and Hong Hu. 2025. Parser Knows Best: Testing DBMS with Coverage-Guided Grammar-Rule Traversal. *arXiv preprint arXiv:2503.03893* (2025).

[26] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting logical bugs of {DBMS} with coverage-based guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. 4309–4326.

[27] Li Lin, Zongyin Hao, Chengpeng Wang, Zhuangda Wang, Rongxin Wu, and Gang Fan. 2024. SQLess: Dialect-Agnostic SQL Query Simplification. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 743–754.

[28] Priti Mishra and Margaret H Eich. 1992. Join processing in relational databases. *ACM Computing Surveys (CSUR)* 24, 1 (1992), 63–113.

[29] Stratos Idreos Fabian Groffen Niels Nes and Stefan Manegold Sjoerd Mullender Martin Kersten. 2012. MonetDB: Two decades of research in column-oriented database architectures. *Data Engineering* 40 (2012).

[30] Thomas Neumann. 2009. Query simplification: graceful degradation for join-order optimization. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 403–414.

[31] Thomas Neumann. 2025. Improving Unnesting of Complex Queries. In *Datenbanksysteme für Business, Technologie und Web (BTW 2025)*. Gesellschaft für Informatik, Bonn, 25–47.

[32] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*, Vol. 20. 29.

[33] Thorsten Papenbrock and Felix Naumann. 2017. Data-driven Schema Normalization.. In *EDBT*, Vol. 17. 342–353.

[34] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science-Towards Embedded Analytics.. In *CIDR*.

[35] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1140–1152.

[36] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[37] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 667–682.

[38] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. 23–34.

[39] Kostya Serebryany. 2017. {OSS-Fuzz}-Google's continuous fuzzing service for open source software. (2017).

[40] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing database systems via differential query execution. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2072–2084.

[41] Michael Stillger and Johann Christoph Freytag. 1995. Testing the quality of a query optimizer. *IEEE Data Eng. Bull.* 18, 3 (1995), 41–48.

[42] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1493–1509.

[43] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making transactional {key-value} stores verifiably serializable. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 63–80.

[44] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting logic bugs of join optimizations in dbms. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[45] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on software engineering* 28, 2 (2002), 183–200.
[46] Chi Zhang and Manuel Rigger. 2025. Constant Optimization Driven Database System Testing. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–24.
[47] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 955–970.
[48] Suyang Zhong and Manuel Rigger. 2025. Testing Database Systems with Large Language Model Synthesized Fragments. *arXiv preprint arXiv:2505.02012* (2025).
[49] Suyang Zhong and Manuel Rigger. 2026. Scaling Automated Database System Testing. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (USA) *(ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 1677–1692. doi:10.1145/3779212.3790215
[50] Xiaogang Zhu and Marcel Böhme. 2021. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2169–2182.

Received October 2025; revised January 2026; accepted February 2026