Finding Logic Bugs in Spatial Database Engines via *Affine Equivalent Inputs*

WENJING DENG, East China Normal University, China
QIUYANG MANG, The Chinese University of Hong Kong, Shenzhen, China
CHENGYU ZHANG, ETH Zurich, Switzerland
MANUEL RIGGER, National University of Singapore, Singapore

Spatial Database Management Systems (SDBMSs) aim to store, manipulate, and retrieve *spatial data*. SDBMSs are employed in various modern applications, such as geographic information systems, computer-aided design tools, and location-based services. However, the presence of logic bugs in SDBMSs can lead to incorrect results, substantially undermining the reliability of these applications. Detecting logic bugs in SDBMSs is challenging due to the lack of ground truth for identifying incorrect results. In this paper, we propose an automated *geometry-aware generator* to generate high-quality SQL statements for SDBMSs and a novel concept named *Affine Equivalent Inputs* (*AEI*) to validate the results of SDBMSs. We implemented them as a tool named Spatter (Spatial DBMSs Tester) for finding logic bugs in four popular SDBMSs: PostGIS, DuckDB Spatial, MySQL, and SQL Server. Our testing campaign detected 34 previously unknown and unique bugs in these SDBMS, of which 30 have been confirmed, and 18 have been already fixed. Our testing efforts have been well appreciated by the developers. Experimental results demonstrate that the *geometry-aware generator* significantly outperforms a naive random-shape generator in detecting unique bugs, and *AEI* can identify 14 logic bugs in SDBMSs that were overlooked by previous methodologies.

CCS Concepts: • **Information systems → Data management systems**; • **Software and its engineering → Software testing and debugging**.

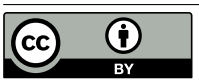
Additional Key Words and Phrases: Spatial query processing, logic bug

ACM Reference Format:
Wenjing Deng, Qiuyang Mang, Chengyu Zhang, and Manuel Rigger. 2024. Finding Logic Bugs in Spatial Database Engines via *Affine Equivalent Inputs*. *Proc. ACM Manag. Data* 2, 6 (SIGMOD), Article 235 (December 2024), 26 pages. https://doi.org/10.1145/3698810

1 Introduction

Spatial Database Management Systems (SDBMSs) aim to store, manipulate, and retrieve *spatial data*, which describes objects and locations under a coordinate system [15]. SDBMSs are employed in various modern applications, such as geographic information systems [34], computer-aided design [23], location-based services [47], and scientific simulations [27, 39]. SDBMSs are typically implemented as spatial extensions or build-in features of widely-used relational DBMSs. For example, PostGIS, the most popular SDBMS on DB-Engines Ranking, is a spatial extension of PostgreSQL; MySQL, one of the most widely-used relational DBMSs supports spatial built-in features.

Authors' Contact Information: Wenjing Deng, East China Normal University, Shanghai, China, 51215902117@stu.ecnu.edu.cn; Qiuyang Mang, The Chinese University of Hong Kong, Shenzhen, Guangdong, China, qiuyangmang@link.cuhk.edu.cn; Chengyu Zhang, ETH Zurich, Zurich, Switzerland, chengyu.zhang@inf.ethz.ch; Manuel Rigger, National University of Singapore, Singapore, Singapore, rigger@nus.edu.sg.



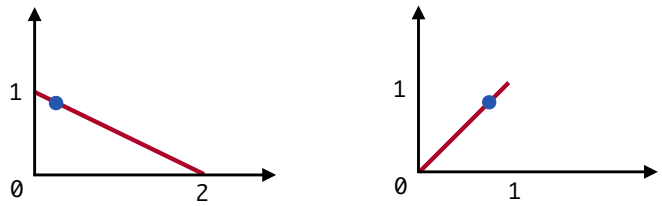
ACM 2836-6573/2024/12-ART235
https://doi.org/10.1145/3698810

Listing 1. Statements that trigger a bug in PostGIS. The retrieved value from PostGIS should be 1 instead of 0.

```
1 CREATE TABLE t1 (g geometry);
2 CREATE TABLE t2 (g geometry);
3 INSERT INTO t1 (g) VALUES ('LINESTRING(0 1,2 0)');
4 INSERT INTO t2 (g) VALUES ('POINT(0.2 0.9)');
5 SELECT COUNT(*) FROM t1 JOIN t2 ON ST_Covers(t1.g,t2.g);
6 -- {0}  {1}
```

Listing 2. The statements correspond to the geometries in Figure 1(b).

```
1 CREATE TABLE t1 (g geometry);
2 CREATE TABLE t2 (g geometry);
3 INSERT INTO t1 (g) VALUES ('LINESTRING(1 1,0 0)');
4 INSERT INTO t2 (g) VALUES ('POINT(0.9 0.9)');
5 SELECT COUNT(*) FROM t1 JOIN t2 ON ST_Covers(t1.g,t2.g);
6 -- {1}
```



(a) The line covers the point.

(b) Geometries affine transformed from left geometries.

Fig. 1. Visualizations of the geometries in Listing 1 and Listing 2 are shown in (a) and (b) respectively. The geometry pairs in (a) and (b) are affine equivalent.

Despite the importance of SDBMSs, their reliability has not received sufficient attention. While general-purpose fuzzers have been applied to SDBMSs for generating obscure inputs that cause crashes [32]—namely crash bug detection—they have failed to detect logic bugs. Logic bugs are a particularly notorious category of bugs, causing the SDBMSs to compute incorrect results. Unlike crash bugs, which terminate the process, logic bugs silently produce incorrect results and thus often go unnoticed by both developers and application users.

Listing 1 shows statements that trigger a logic bug in PostGIS. The statements insert a line segment and a point into the database (Line 1–4) and query whether the line segment covers the point (Line 5). The line segment and the point are drawn in Figure 1(a), clearly showing that the line covers the point, which means the result should be `1`. However, PostGIS incorrectly returned `0`, which indicates a logic bug.

Detecting logic bugs automatically in SDBMSs remains a challenging problem. To the best of our knowledge, in SDBMSs, past practices largely rely on user reports to identify logic bugs; no automated testing strategies have been applied. The key challenge of automatically finding logic bugs in SDBMSs is the lack of ground truth results. Many automated testing techniques have been proposed for detecting logic bugs in relational DBMSs [1, 8, 20, 25, 35–38], but unfortunately, they are not applicable for testing SDBMSs, especially for spatial-related features.

One methodology of testing relational DBMS works is to generate the query, pass it to different systems, and consider the equivalence of their outputs as the expected result, a technique known as *differential testing* [8, 20]. However, for spatial-related features that are solely implemented in one SDBMS, differential testing techniques are inapplicable, because the expected result cannot be constructed. In addition, bugs can also be in the shared third-party libraries, leading multiple SDBMSs to produce consistent, but incorrect outputs, resulting in bugs being missed. Furthermore, although the *Open Geospatial Consortium Standards* (OGC) have standardized most functions and data types in SDBMSs, implementation details still vary among SDBMSs [31], which means their outputs are expected to be nonequivalent, resulting in *false alarms*. We observed various expected discrepancies between SDBMSs—as per developers' intents. The bug in Listing 1 cannot be detected by differential testing, as function `ST_Covers` is only implemented in PostGIS and DuckDB Spatial, relying on their common third-party library.

Another methodology involves generating statements and constructing their expected results within the same DBMS, thus avoiding the limitations of differential testing. For instance, Ternary Logic Partitioning (TLP) is a general state-of-the-art testing technique for relational DBMSs [36], applicable not only to relational DBMSs, but also to graph DBMSs [21]. The key insight of TLP is to derive three partitioning queries from an original query, and the sum of the results from the partitioned queries is expected to be the same as that of the original one. However, TLP may fail to detect logic bugs in spatial-related features. For example, the bug presented in Listing 1 cannot be found by TLP because both the summed-up results of the partitioning queries and the original query result are incorrect. Therefore, an SDBMS-oriented approach for generating test cases and expected results is necessary.

We propose a methodology named *Affine Equivalent Inputs* (*AEI*) to provide the expected results for SDBMSs. Our key insight is that if two geometries affine transform (*e.g.*, rotate, scale, and translate) in the same way, *topological relationships* (*e.g.*, intersects, covers, or disjoint) are preserved. Therefore, two topological relationships of two *affine equivalent* geometry pairs retrieved from an SDBMS are expected to be equal; otherwise, a bug is detected. We consider two geometry pairs to be affine equivalent if they can be transformed into each other through an affine transformation.

For example, considering the geometries in Figure 1(a), affine transforming them yields the geometries in Figure 1(b) corresponding to the statements in Listing 2. Since affine transformations are invertible [5], the geometry pairs in Figure 1(a) and Figure 1(b) are deemed *affine equivalent*. With the expectation of equal topological relationships, the statements in Listing 1 and 2 should yield the same results. However, PostGIS correctly evaluates `1` for the statements in Listing 2 and incorrectly returns `0` for the statements in Listing 1, revealing a logic bug. We found this bug, which was caused by precision issues, via *AEI*. Specifically, the bug was caused by a loss of precision in the normalization of vertices (*i.e.*, displacement to the origin). The statements in Listing 2 fail to trigger this bug, because a point in v is at the origin, leading to no displacement calculations. Overall, in this example, *AEI* reveals the bug by triggering different execution paths (*i.e.*, with/without displacement calculations). As demonstrated through the above example, such precision issues may result in inaccuracies in topological relationship calculations. The developers of PostGIS have acknowledged this bug and have highlighted their concern by modifying the issue title to *"Covers predicate fails on obviously correct simple case"*. After we reported this bug, the developers indicated that they would start working on a new predicate evaluation mechanism, which would include a tolerance argument to prevent such issues.

Based on *AEI*, we implemented a tool named Spatter for testing SDBMSs.[1] Spatter consists of a *geometry-aware generator* for generating high-quality SQL statements for SDBMSs and uses *AEI*

[1]Our artifact is publicly available at https://github.com/cuteDen-ECNU/Spatter and https://zenodo.org/records/13932460.

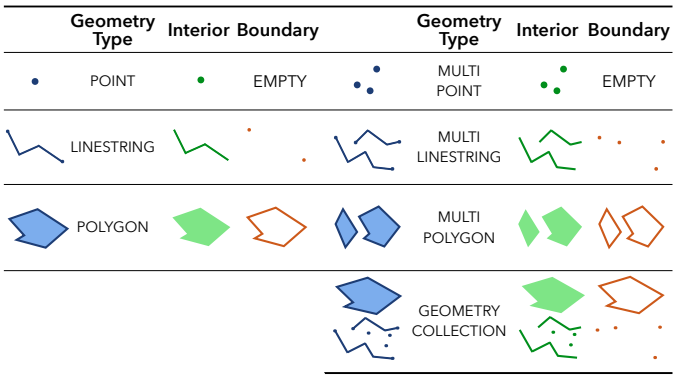


Fig. 2. Examples of 2D geometries and their geometry types, interiors, and boundaries.

to validate the results of SDBMSs. To the best of our knowledge, Spatter is the first automated testing tool to detect logic bugs in SDBMSs. To evaluate the effectiveness of *AEI*, we selected 4 widely-used SDBMSs as our testing targets: PostGIS, MySQL, DuckDB Spatial, and SQL Server. The results show that our approach is effective in detecting spatial-related logic bugs in SDBMS. Our testing campaign detected 34 real, previously unknown, and unique bugs, which were missed by existing test suites. 30 of them have been confirmed, 18 have been fixed, and 20 were logic bugs. During the testing campaign, our work received positive feedback from the SDBMSs' developers. Our experimental results demonstrate that the *geometry-aware generator* significantly outperforms a naive random-shape generator in detecting unique bugs, and *AEI* could identify 14 logic bugs in SDBMSs that were totally overlooked by previous methodologies.

To summarize, we make the following contributions:

- We propose *Affine Equivalent Inputs*—a methodology to automatically validate the results of SDBMSs.
- We designed and implemented an automated tool Spatter for detecting logic bugs in SDBMSs.
- We detected 34 previously-unknown and unique bugs in four widely-used SDBMSs.
- We evaluated Spatter and show that it outperforms the baseline strategies.

2 Background

In this section, we provide important background information on geometry types, topological relationship queries, and affine transformations.

2.1 Geometry Types

SDBMSs provide geometry types, on which we focus, given that they are widely used in real-world datasets [11] and support various functions. Figure 2 illustrates seven widely-used 2D geometry types; these and other geometry types are standardized by *Open Geospatial Consortium Standards* (OGC) [28]. The basic geometry types shown on the left include POINT, LINESTRING, and POLYGON, with dimensions of 0, 1, and 2 respectively. A geometry whose type starts with "MULTI" consists of multiple elements of the same basic type; we refer to this as *MULTI geometry*. The type GEOMETRYCOLLECTION comprises elements of mixed geometry type; we refer to this as *MIXED geometry*. Examples of *MULTI* and *MIXED* geometries are shown on the right side of Figure 2.

2.2 Topological Relationship Queries

Topological relationships represent qualitative properties that characterize the relative position of spatial objects [12]. The concept of a *topological relationship query* is a core feature when operating on spatial data, playing a key role in spatial queries and joins. Therefore, the correctness of the execution engine in a SDBMS that processes such topological relationship queries is crucial. Despite this, to the best of our knowledge, no automated testing approach for detecting logic bugs in such functionality has been proposed. *Formal topological relationships* and *named topological relationships* are two types of widely-used topological relationships in SDBMSs.

Definition of formal topological relationships. *Formal topological relationships* are formally defined in the Dimensionally Extended 9-Intersection Model (DE-9IM [6, 7]), which establishes spatial relationships between two given geometries.

In DE-9IM, the geometry is conceptualized as a *cell complex* using principles from algebraic topology [9]. A cell complex consists of cells and their respective *faces*. The faces of a cell are crucial components that delineate the cell's boundaries. [10]. Cells are primitive geometric objects defined across various spatial dimensions. Specifically, an n-cell is composed of $n+1$ geometrically independent cells of dimension $n-1$ [10]. A *face* of n-cell C is any r-cell contained within C, where $0 \leqslant r \leqslant n$ [10].

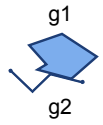
Examples of cells are 0-cells representing points, 1-cells for edges, 2-cells for triangles, and 3-cells for tetrahedrons. For example, a 2-cell has *faces* that include two *1-cells* (edges) and one *2-cell*, representing its end-points and the edge connecting these end-points respectively. In SDBMS implementations, a `POINT` is a 0-cell; a `LINESTRING` is described as a cell complex composed of multiple 1-cells and their faces; a `POLYGON` is represented as a cell complex consisting of several 2-cells and their faces.

Formal topological relationships are qualitatively defined based on the concepts of cell complexes' boundary, interior, and exterior. Firstly, we precisely present the definition of *boundary*, *interior*, and *exterior* for cells in Definition 2.1. For instance, in a triangle (*i.e.*, a 2-cell), the boundary consists of its three vertices and three edges; the interior is the area it encloses; the exterior is defined as the set from which the boundary and interior are excluded. On top of Definition 2.1, Definition 2.2 further elucidates the concepts of boundary, interior, and exterior within cell complexes.

Definition 2.1 (Boundary, Interior, and Exterior of Cells). The closure of an n-cell C, denoted by $\overline{C}$, is the set of all *r-faces* $f(r)$ of C, where $0 \leqslant r \leqslant n$, that is, $\overline{C} = \bigcup_{r=0}^{n} f(r)$. The boundary of C, denoted by ∂C, is a finite union set of r-faces of C, where $0 \leqslant r \leqslant (n-1)$, that is, $\partial C = \bigcup_{r=0}^{n-1} f(r)$. The interior of a cell C, denoted by C^o, is the difference set of closure and boundary of C, that is, $C^o = \overline{C} - \partial C$. The exterior of a cell C, denoted by C^-, is the difference set of universal set $\mathbb{U}$ and closure of C, that is, $C^- = \mathbb{U} - \overline{C}$ [10].

Definition 2.2 (Boundary, Interior, Exterior of Cell Complexes). Let x be the number of cells $(C_1 ... C_x)$ that constitute a cell complex G. The boundary of G, denoted by ∂G, is the union of all the boundaries of C_i, while subtracting the intersections between boundaries of any distinct cell pairs, that is, $\partial G = \bigcup_{i=1}^{x} \partial C - \bigcup_{i=1}^{x} \bigcup_{j=i+1}^{x} (\partial C_i \cap \partial C_j)$. The interior of G, denoted as G^o, is the union set of the closure of C (*i.e.*, $\overline{C}$), subtracting ∂G, that is, $G^o = \bigcup_{i=1}^{x} \overline{C_i} - \partial G$. The exterior of G is the intersection of C_i^-, that is, $G^- = \bigcap_{i=1}^{x} C_i^-$ [10].

Figure 2 intuitively demonstrates cell complexes, namely geometries as well as their interiors and boundaries. Specifically, the boundary of POINT is defined as an empty set. The LINESTRING in the figure is a cell complex composed of three connected line segments (1-cells), with its boundary



DE-9IM	Interior(g2)	Boundary(g2)	Exterior(g2)
Interior(g1)	F	F	2
Boundary(g1)	1	F	1
Exterior(g1)	1	0	2

Fig. 3. DE-9IM code of POLYGON g1 and LINESTRING g2 is FF21F1102. The letter F indicates that the intersection is an empty set. The numbers 0, 1, and 2 represent the dimension of the intersection.

defined by its two endpoints. The POLYGON in the figure is a cell complex consisting of multiple triangles, with its boundary including its vertices and edges.

DE-9IM qualitatively defines and determines the *topological relationship* between two spatial entities. It does so by characterizing the relationships between two cell complexes through the interactions among their boundaries (denoted by g^o), interiors (denoted by ∂g), and exteriors (denoted by g^-). The model employs a 3×3 matrix, which is constructed with the aid of a dimension calculator, denoted as $\mathcal{D}$. When the intersection between the compared elements is empty, the calculator yields a constant value, F; otherwise, it returns n, a value that represents the dimension of the intersection.

Definition 2.3 (Formal Topological Relationship). For any two cell complexes g_1 and g_2,

$$R(g_1, g_2) = \begin{bmatrix} \mathcal{D}[g_1^o \cap g_2^o] & \mathcal{D}[g_1^o \cap \partial g_2] & \mathcal{D}[g_1^o \cap g_2^-] \\ \mathcal{D}[\partial g_1 \cap g_2^o] & \mathcal{D}[\partial g_1 \cap \partial g_2] & \mathcal{D}[\partial g_1 \cap g_2^-] \\ \mathcal{D}[g_1^- \cap g_2^o] & \mathcal{D}[g_1^- \cap \partial g_2] & \mathcal{D}[g_1^- \cap g_2^-] \end{bmatrix}, \tag{1}$$

where $\mathcal{D}$ is a dimension calculator.

Figure 3 illustrates the *formal topological relationship* between two example geometries $g1$ and $g2$. For instance, the dimension of the intersection between the boundary of g1 and the interior of g2 is 1 (*i.e.*, $\partial g_1 \cap g_2^o = 1$), that is, the boundary of g1 and the interior of g2 intersect along an edge of dimension 1. Furthermore, the intersection of the interiors of g1 and g2 is an empty set (*i.e.*, $g_1^o \cap g_2^o = F$), indicating that the interior of $g1$ does not intersect with the interior of $g2$. In PostGIS, the function `ST_Relate(g1,g2)`, which is implemented based on DE-9IM, evaluates the formal topological relationship between $g1$ and $g2$. This relationship is represented as a string of length 9 with a domain of $\{0, 1, 2, F\}$. Specifically, the DE-9IM code of g1 and g2 is `FF21F1102`.

Named topological relationships. *Named topological relationships* are derived from *formal topological relationships*. Unlike *formal topological relationships*, they describe relationships in a manner that is easily understood by users [6]. A set of named topological relationships, such as functions `ST_Disjoint` and `ST_Intersects` are widely supported by SDBMSs. Besides, SDBMSs extend their functionality through specific named topological relationship queries. For example, as seen in Listing 1, the function `ST_Cover` is only implemented in PostGIS and DuckDB Spatial.

Given that well-defined *formal topological relationships* and human-readable *named topological relationship* queries are widely used in various scenarios [41], ensuring their correctness is crucial. As far as we know, no general testing approach exists to identify logic bugs in queries regarding topological relationships, despite their importance.

2.3 Affine Transformations

We explore a fundamental concept in mathematics and computer graphics known as *affine transformation*. Our intuition is that *affine transformations* maintain topological relationships, because they preserve topological properties within *Euclidean spaces* [5]. Figure 4 presents intuitive examples

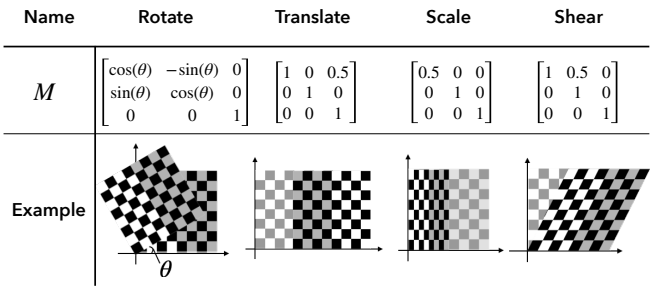


Fig. 4. Examples of affine transformation.

of *affine transformations*, including *rotation*, *translation*, *scaling*, and *shearing*. The checkerboard patterns, differentiated by high and low transparency, illustrate the geometries before and after the transformations, respectively.

The widely-used topological relationship queries are implemented in *Euclidean spaces* $\mathbb{R}^2$ and $\mathbb{R}^3$ in SDBMS. An affine transformation $\mathcal{A} : \mathbb{R}^2 \to \mathbb{R}^2$ for a point $p \in \mathbb{R}^2$ is defined as

$$\mathcal{A}(p) = \mathbf{A}p + \mathbf{b} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}, \tag{2}$$

and $\mathcal{A} : \mathbb{R}^3 \to \mathbb{R}^3$ for any point $p \in \mathbb{R}^3$ is defined as

$$\mathcal{A}(p) = \mathbf{A}p + \mathbf{b} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}, \tag{3}$$

where $\mathbf{A}$ denotes an invertible matrix that describe the linear transformation; and $\mathbf{b}$ presents the translate vector.

The mapping matrix M in Figure 4 is an augmented matrix that represents both the linear transformation and translation. M has one more row and column than Equation (2) and (3):

$$\alpha' = M\alpha = \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{0} & 1 \end{bmatrix} \alpha, \tag{4}$$

where α is the homogeneous vector of point p.

3 Affine Equivalent Inputs

In this section, we present the methodology to construct the expected result. First, we formalize our intuition that *topological relationships* remain preserved after applying *affine transformations*. Then, we provide a proof of this proposition. Based on the proved proposition, we define *Affine Equivalent Inputs* (*AEI*).

Proof of our intuition. To formalize our intuition, we propose Proposition 3.3, asserting that the topological relationship between a geometry pair is equal to that between its *affine equivalent* pair. We use *formal topological relationships* definition in Equation 1, since *named topological relationships* are derived from them. In *formal topological relationships*, geometries are characterized by *cell complexes*. Given that the queries regarding topological relationships for geometry types are executed within the *Euclidean spaces* $\mathbb{R}^2$ and $\mathbb{R}^3$, we contextualize these *cell complexes* accordingly.

We introduce the concept of *Euclidean spaces*, denoted as $\mathbb{R}^n$ where $2 \leq n \leq 3$, as a framework for delineating the structure of cell complexes. A geometry can be defined by a cell complex composed of cells. Therefore, prior to defining affine equivalent geometries, we first define affine transformations

on cells. A 0-cell (*i.e.*, a point) in space $\mathbb{R}^2$ and $\mathbb{R}^3$ can be represented using Euclidean coordinates (x, y) and (x, y, z) respectively. A 1-cell, corresponding to a line segment, is characterized by the coordinates of its two distinct endpoints; a 2-cell, representing a triangle, is defined by three non-collinear edges; a 3-cell, or a tetrahedron, consists of four triangles, forming a three-dimensional figure. In $\mathbb{R}^2$ and $\mathbb{R}^3$, 2-cells and 3-cells are the highest-dimension cells, respectively. Note that the geometry discussed here does not include curves. Hence, an n-cell can be described by specifying the coordinates of its vertices. Next, we define the affine transformation of cells in Definition 3.1.

Definition 3.1 (Affine Transformations on Cells). For the vertices $\{p_1, ..., p_n\}$ of a cell, an affine transformation can be defined as $\mathcal{A} : p_i \rightarrow p_i{}'$, where $1 \leq i \leq n$.

To formalize the concept of *affine equivalent geometries*, we define an *affine equivalence triplet* $(g, g', \mathcal{A})$, where g and g' are geometries, and $\mathcal{A}$ represents an affine transformation. We say that g and g' are affine equivalent under $\mathcal{A}$ if two conditions are both satisfied: (1) for any cell c in g, there exists a corresponding cell c' in g' such that $\mathcal{A}$ maps c to c', and (2) for any cell c' in g', there exists a corresponding cell c in g such that the inverse of $\mathcal{A}$ maps c' back to c. Specifically, we apply an affine transformation to a geometry to generate a pair of *affine equivalent geometries*. We further clarify the definition of affine equivalent geometry pairs in Definition 3.2. Subsequently, in Proposition 3.3, we formalize and prove our premise that the formal topological relationships remain invariant in pairs of affine equivalent geometries.

Definition 3.2 (Affine Equivalent Geometry Pairs). Considering two geometry ordered pairs (g_1, g_2) and (g_1', g_2'), they are affine equivalent geometry pairs if two affine equivalence triplets (g_1, g_1', A) and (g_2, g_2', A) exist.

Proposition 3.3. *For a geometry pair (g_1, g_2), and its affine equivalent geometry pair (g_1', g_2'), we have $R(g_1, g_2) = R(g_1', g_2')$.*

Proof. For brevity, we show the correctness of the relation between *Boundary* of g_1 and *Boundary* of g_2, *i.e.*, $\mathcal{D}[\partial g_1 \cap \partial g_2] = \mathcal{D}[\partial g_1' \cap \partial g_2']$, while omitting the proof for remaining relations in R that can be proven similarly. To this end, we carefully consider the only two possible intersection cases for ∂g_1 and ∂g_2: *non-empty* and *empty* set.

(1) ***Non-empty set.*** In this case, the intersection of ∂g_1 and ∂g_2 is a non-empty set. Suppose k cells in the set, denoted as $\partial g_1 \cap \partial g_2 = \sigma : \{\sigma_1, \ldots, \sigma_k\}$. According to the Definition 3.2, for any cell $\sigma_i (1 \leq i \leq k)$, there exists a cell σ_i' belonging to both $\partial g_1'$ and $\partial g_2'$, which implies $\sigma_i' \in \partial g_1' \cap \partial g_2'$ and thus $\mathcal{A} : \sigma \rightarrow \sigma'$ holds. Similarly, we have $\mathcal{A}^{-1} : \sigma' \rightarrow \sigma$. Therefore, $\partial g_1 \cap \partial g_2$ is bijective to $\partial g_1' \cap \partial g_2'$. Considering that *affine transformation* preserves the dimension [5], $\mathcal{D}[\partial g_1 \cap \partial g_2] = \mathcal{D}[\partial g_1' \cap \partial g_2']$ is thus proved in the non-empty set case.

(2) ***Empty set.*** In this case, the intersection of ∂g_1 and ∂g_2 is an empty set. We prove that the intersection of $\partial g_1'$ and $\partial g_2'$ is also an empty set by contradiction. Assuming $\partial g_1' \cap \partial g_2' \neq \emptyset$, there must exist a face σ_p in $\partial g_1' \cap \partial g_2'$. According to the definition of affine equivalent geometry pairs, we can also find a face $\sigma_p' \in \partial g_1 \cap \partial g_2$ by $\mathcal{A}^{-1} : \sigma' \rightarrow \sigma$, which contradicts $\partial g_1 \cap \partial g_2 = \emptyset$.

□

***Definition of* AEI**. We define a triplet $I = (\alpha, \beta, Q)$ as an input for an SDBMS, where α and β are two geometries, and Q is a topological relationship query on geometry pair α and β.

*Definition 3.4 (*Affine Equivalent Inputs*).* $I_1 = (\alpha_1, \beta_1, Q_1)$ and $I_2 = (\alpha_2, \beta_2, Q_2)$ are *Affine Equivalent Inputs*, if geometry pairs (α_1, β_1) and (α_2, β_2) are affine equivalent and Q_1 is equal to Q_2.

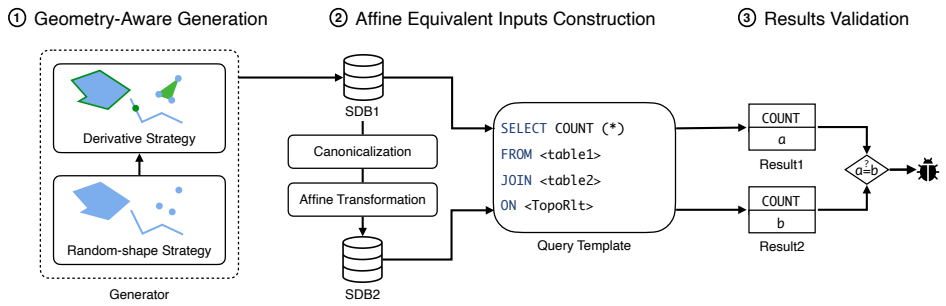


Fig. 5. Overview of Spatter

According to Proposition 3.3, passing *Affine Equivalent Inputs* to the same SDBMS should cause the topological relationship query to return the same result, unless the SDBMS is affected by a bug.

4 Spatter

We propose Spatter (Spatial DBMSs Tester), an automated testing tool and approach that combines *AEI* with a *geometry-aware generator*. The core idea behind *AEI* is to construct two *affine equivalent* geometry pairs and check the consistency of their topological relationships. The *geometry-aware generator* introduces a new concept for efficiently generating interesting test inputs, which not only generates random geometries (using a *random-shape strategy*), but also derives geometries by applying spatial functions to existing geometries (using a *derivative strategy*). The key insight behind the *geometry-aware generator* is its ability to generate various topological relationships with a limited number of geometries, thereby enabling *AEI* to validate the results efficiently.

Figure 5 shows an overview of Spatter. Initially, we create a database SDB1 with N geometries using the *geometry-aware generator*'s *random-shape* and *derivative* strategies (see Step ①); the *derivative strategy* creates geometries based on existing ones. In the next step, we canonicalize each geometry in SDB1 by obtaining an equivalent, potentially equivalent representation of the geometry, and then apply an affine transformation to construct a new geometry, resulting in SDB2 (see Step ②). Since *affine transformations* preserve the topological relationships and *canonicalization* produces equivalent geometries at the spatial level, the set of topological relationships of SDB1 is equivalent to that of SDB2, which we check for result validation. For result validation, we randomly fill the placeholders of a query; then we check whether the row counts retrieved from the same SDBMS by this query are consistent between SDB1 and SDB2. Inconsistent counts indicate a logic bug (see Step ③).

4.1 Geometry-Aware Generator

In this section, we introduce a crucial component of Spatter: a generator designed for efficiently detecting bugs in SDBMSs. Previously, efforts to find logic bugs in SDBMSs relied on unit tests and user reports, as no automated geometry generators were designed for this purpose. One naive approach is to randomly generate syntactically valid geometries, corresponding to our *random-shape strategy*. However, the *random-shape strategy* makes it unlikely to observe a variety of topological relationships for the generated geometries, making it difficult to exercise the SDBMSs. This issue is further exacerbated by the necessity of setting a limit on the number of geometries to avoid excessive time overheads in SDBMSs (see Section 5.4). To improve the efficiency of bug finding, we believe that a wider range of topological relationships within this limited number of geometries is required, allowing us to detect more logic bugs in queries regarding topological

Algorithm 1 Geometry-Aware Generator

Require: N, geometry number; m, table number
Ensure: sdb, generated spatial database
1: **function** Generate(N, m)
2: $sdb \leftarrow$ CreateTables(m)
3: $g \leftarrow$ RandomShape()
4: $sdb \leftarrow$ InsertIntoRandomTable(sdb, g)
5: **for** _ $\leftarrow$ 2 to N **do**
6: **if** RandomShape() is True **then**
7: $g \leftarrow$ RandomShape()
8: **else**
9: $editFunc \leftarrow$ RandomEditFunction()
10: $g \leftarrow$ Derive(sdb, $editFunc$)
11: $sdb \leftarrow$ InsertIntoRandomTable(sdb, g)
12: **return** sdb
13: **function** RandomShape( )
14: $gType \leftarrow$ RandomGeometryType()
15: $g \leftarrow$ SyntaxGen(gType)
16: **return** g
17: **function** Derive(sdb, $editFunc$)
18: $k \leftarrow$ the geometry number $editFunc$ needed
19: $gVector \leftarrow$ k geometries randomly selected from sdb
20: $g \leftarrow$ editFunc($gVector$)
21: **if** Deriving g failed **then**
22: $g \leftarrow$ EmptyShape()
23: **return** g

relationships. Therefore, we propose the *derivative strategy* that derives existing geometries by applying spatial functions. The *geometry-aware generator* employs two strategies for generating geometries: the *random-shape* and *derivative* strategies.

Random-shape strategy. The *random-shape strategy* generates syntactically valid geometries. Line 13–16 in Algorithm 1 demonstrate how the *random-shape strategy* generates a geometry: first, it randomly selects a geometry type $gType$ (Line 14), and then uses the syntax definition for $gType$ to generate the geometry g (Line 15). Geometries created using the *random-shape strategy* are valid at the syntax level, but not necessarily at the semantic level. For example, the polygon `POLYGON((0 0,1 1,0 1,1 0,0 0))` is an invalid shape due to its self-intersecting boundaries, but it can be generated by the *random-shape strategy* since it is syntactically valid. When a geometry is semantically invalid, the SDBMS will indicate this with an error, which we ignore in Spatter.

Derivative strategy. The *derivative strategy* is designed to create more topological relationships by deriving a geometry based on a set of editing functions provided by SDBMSs. Line 17–23 in Algorithm 1 illustrate how the *derivative strategy* generates a geometry. First, we obtain the input number k of editing function $editFunc$ (Line 18). Then, k geometries are randomly selected from sdb to construct a geometry vector $gVector$ (Line 19). The new geometry g is derived from $gVector$ by $editFunc$ (Line 20). However, the derivation process may encounter errors if the $editFunc$ does not apply to the geometries in $gVector$. To address this, we check for any failures, opting to generate an *EMPTY* geometry in case of failure (Line 21–22). Finally, it returns geometry g derived from the existing k geometries.

Table 1. Categories of the operations in the *derivative strategy*.

Type	Example	Description
Line-Based	SetPoint	Replace a specific point of an input LINESTRING with a given point.
	Polygonize	Create a GEOMETRYCOLLECTION containing the polygons formed by the line.
Polygon-Based	DumpRings	Extract the rings of an input POLYGON.
	ForcePolygonCW	Force an input POLYGON or MULTIPOLYGON to use a clockwise orientation for their exterior ring.
Multi-Dimensional	GeometryN	Fetch the Nth geometry element from an input MULTI or MIXED geometry based on 1-based indexing.
	CollectionExtract	Produce a collection of geometries of a specified type, extracted from an input MULTI or MIXED geometry.
Generic	Boundary	Retrieve the boundary of an input geometry.
	ConvexHull	Generate the convex hull of an input geometry.

Table 1 shows editing functions categorized by geometric dimensions of input geometries. Generic functions can handle geometries of any dimension. In contrast, dimensional-based functions such as line-based, polygon-based, and multi-dimensional functions, operate exclusively on lines, polygons, and *MULTI* and *MIXED* geometries, respectively.

Generation process. The *Generate* function in Algorithm 1 outlines the generation process of the *geometry-aware generator*. The algorithm expects a specified number of geometries, N, and a target number of tables, m, as input. During the initialization phase, a spatial database *sdb* is created with m empty tables (Line 2). Since no geometries can be derived from an empty table, we employ the *random-shape strategy* to generate a geometry and randomly insert it into one of *sdb* tables (Line 3). For the creation of each geometry g, we randomly select one strategy and insert the generated geometry into a randomly chosen table (Line 5–11). Last, we return the newly-created spatial database *sdb*.

4.2 Affine Transformation

Proposition 3.3 suggests a high-level idea: *affine transformations* preserve topological relationships. Therefore, we apply the same affine transformation to each geometry in the generated database.

Algorithm 2 shows the workflow of constructing an *affine equivalent* geometry for g. The inputs of this workflow are the geometry g and the *Euclidean space* dimension n. We first generate a mapping matrix M (Line 2). The *GenerateMappingMatrix* function involves generating an invertible matrix A and a translation vector b (Line 7–11). Then, we create g', a copy of the geometry g (Line 3). For each point p in g', we update its coordinates with the values generated by the *Affine* function (Line 4–5). The *Affine* function performs three steps: converting the coordinate point p into a homogeneous vector, left-multiplying the vector by matrix M, referring to Equation (4), and transforming the resulting vector back into a coordinate point. Finally, the algorithm returns g', which is an affine equivalent geometry of g (Line 6).

Avoiding precision issues. An affine transformation utilizes a matrix to construct affine equivalent geometries. However, using matrices with floating-point numbers introduces precision issues. For example, consider Equation (5), where we apply an affine transformation to a 2D point represented by `POINT(0.1,0)`. We first convert the point into a homogeneous vector p, then generate a matrix M to left-multiply p, obtaining a vector p', and finally convert it back to a point with coordinates $(0.02 + 4 \times 10^{-18}, 0)$. Precision issues arise because multiplying floating-point numbers on computers yields values that are very close to, but not exactly, the theoretical values. Specifically, while in theory, 0.1 multiplied by 0.2 equals 0.02, the representation of floating-point numbers in computers may lead to an imprecise value, with a potential error of `4E-18`.

$$p' = M \cdot p = \begin{bmatrix} 0.2 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.02 + 4 \times 10^{-18} \\ 0 \\ 1 \end{bmatrix} \tag{5}$$

Algorithm 2 Apply an affine transformation on a geometry.

Require: g, a geometry; n, Euclidean space dimension
Ensure: g', affine equivalent input of g
1: **function** Construct(g, n)
2: $M \leftarrow$ GenerateMappingMatrix(n)
3: $g' \leftarrow g$
4: **for each** p **in** g' **do**
5: $p \leftarrow$ Affine(p, M)
6: **return** g'
7: **function** GenerateMappingMatrix(n)
8: $A \leftarrow$ a random non-singular matrix $\in \mathbb{R}^{n \times n}$
9: $b \leftarrow$ a random vector $\in \mathbb{R}^n$
10: $M \leftarrow$ AugmentedMatrix(A, b)
11: **return** M

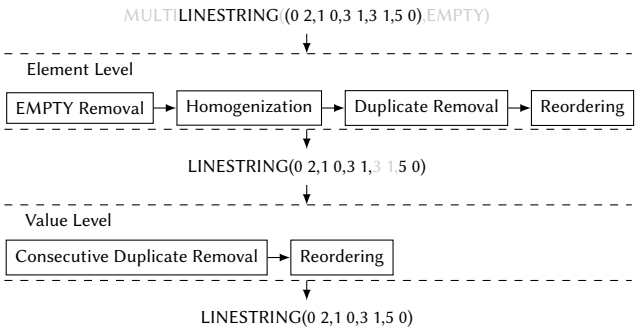


Fig. 6. Canonicalization for geometry at different levels.

Although such errors can be circumvented at the implementation level during affine transformations, precision issues with floating-point numbers still occur in SDBMSs. The bug shown in Listing 1 illustrates an example of precision issues in PostGIS. Despite this bug being confirmed by the developers at the beginning of our testing campaign, bugs unrelated to precision issues are more eagerly anticipated by both the developers and us. To avoid false alarms caused by precision issues, we chose not to introduce floating-point numbers during the steps of affine transformations and spatial data generation. Specifically, for an affine transformation, we generate matrix A (Line 8) and vector b (Line 9) using random integers, as outlined in Algorithm 2.

4.3 Canonicalization

We consider canonicalization as a special case of constructing affine equivalent geometries, since applying the special matrix **E** produces the geometry itself. In canonicalization, we aim to transform the original representation of each geometry into a canonical representation, which not only constructs another kind of expected results effective for testing, but also provides a pre-processing method for Algorithm 2. We define operations to convert the representation of each geometry to a canonical one. We convert the representations at the element and value level.

Element level. The concept of *element* pertains to both MULTI and MIXED geometries. Canonicalization at the element level involves four steps and applies exclusively to MULTI and MIXED geometries. The process begins with *EMPTY removal*, eliminating any *EMPTY* elements present.

Next, *homogenization* transforms a MULTI geometry containing only a single element into a basic-type geometry and flattens nested collections into a uniform structural representation. Subsequently, duplicated elements are removed; note that duplicates are identified based on their shape. Finally, the elements are reordered according to their dimensions.

Figure 6 illustrates the canonicalization of a MULTILINESTRING at the element level. Initially, the *EMPTY* element is removed, yielding a `MULTILINESTRING((2,1 0,3 1,3 1,5 0))`; subsequently, it is homogenized into a `LINESTRING(2,1 0,3 1,3 1,5 0)`, since there is only one element within the MULTI geometry. In this example, neither the *duplicate removal* nor the *reordering* is necessary.

Value level. The value-level canonicalization process is designed for each basic-type element. The location of a basic-type element is specified by an ordered number of pairs or triples. Specifically, in 2D geometries, a POINT is represented by a pair of values; a LINESTRING is presented by ordered points; and a POLYGON is defined by one or multiple loops that ordered points that start and end at the same point.

Canonicalization at the value level involves two steps: *consecutive duplicate removal* and *reordering*. First, we eliminate redundant points that are identical to the preceding point. Then, we reorder the coordinates by direction. For a LINESTRING, we determine whether to reverse by comparing the values of the endpoints in the order of the x-axis, y-axis, and z-axis. For a POLYGON, we convert all the loops into clockwise orientation.

Figure 6 demonstrates the canonicalization of a LINESTRING at the value level. First, we remove the redundant point `(3 1)`, resulting in `LINESTRING(0 2,1 0,3 1,5 0)`. Then, we reorder the coordinates. Since the endpoint coordinates are ordered as expected, reordering does not alter the representation.

4.4 Results Validation

By utilizing *canonicalization* and *affine transformation*, it is possible to establish multiple sets of *Affine Equivalent Inputs* (*AEI*). Consider a spatial database named SDB1. For each geometry g within SDB1, we perform canonicalization and apply an affine transformation to construct its *affine equivalent* geometry, g', resulting in a new spatial database, SDB2. The geometries g and g' are stored in tables of the same name in their respective databases.

Figure 5 presents a query template containing three placeholders: two table names (*i.e.*, <table1>, <table2>) and one for a topological relationship condition (*i.e.*, <TopoRlt>). Two table names are valid and selected randomly from SDB1. The boolean condition <TopoRlt> is valid for the tested SDBMS and is randomly selected from a list containing conditions regarding topological relationships from SDBMS user manuals. When checking whether the results evaluated by the same query are consistent between SDB1 and SDB2, any discrepancy reveals a bug in the tested SDBMSs. To illustrate, consider the case where t1 and t2 are two valid tables in SDB1, and `ST_Covers` is a topological relationship function specific to PostGIS. The query filled with t1, t2, and `ST_Covers (t1.g, t2.g)` retrieves the row count from PostGIS, indicating whether geometries in t1 cover geometries in t2. If the count retrieved from SDB1 is not equal to that from SDB2, we detect a potential logic bug.

5 Evaluation

In this section, we first summarize and classify the detected bugs, along with feedback from developers. Then, we analyze the bugs found by *AEI*, present bug examples, and discuss bug-inducing patterns. Next, we assess whether the previous methodologies could detect the logic identified by *AEI*. Finally, we conduct experiments to demonstrate the efficiency of Spatter.

Tested SDBMSs. We evaluated our approach on 4 well-known, mature, and actively maintained SDBMSs: PostGIS, DuckDB Spatial, MySQL, and SQL Server. PostGIS was chosen as our primary

testing target due to its popularity and high ranking in the DB-Engines Ranking. DuckDB is recognized as the most popular embedded OLAP system [33], and we selected its extension, DuckDB Spatial, for testing. MySQL, one of the most popular open-source relational DBMSs, was tested for its built-in spatial functionality. In addition, we included one commercial SDBMS, namely SQL Server. However, given limited feedback on our reported bugs, we discontinued our testing efforts and did not consider other commercial SDBMSs.

5.1 New Bugs

Methodology. To evaluate the effectiveness of *Affine Equivalent Inputs* in detecting bugs, we incrementally implemented Spatter and intermittently tested the latest versions of SDBMSs over four months in a testing campaign. Typically, Spatter ran for 10 minutes to 1 hour, as it detected issues quickly. Spatter records two sequences of statements for generating the *affine equivalent* databases for each discrepancy. Before reporting any potential issues, we performed two steps. First, we both automatically [45] and manually reduced them. Second, we determined which result violated the definition of topological relationship functions according to the SDBMSs' documentation.

In testing PostGIS and DuckDB Spatial, we discovered that some bugs originated from a common third-party library named GEOS. At the beginning of our testing campaign, we reported potential bugs to the PostGIS and DuckDB Spatial communities. The developers then confirmed them as upstream bugs and reported them to the GEOS community. Subsequently, if we determined the bug was within their library, we directly reported bugs to GEOS. After fixes were applied, we updated the GEOS version on the SDBMSs on our machine to check if the issue persisted. We reported an issue as a separate bug to the SDBMS community only if persisted in the tested SDBMS after all fixes were applied.

Overall bug detection results. Table 2 summarizes the bugs identified during our testing campaign. We classified the detected bugs into four distinct categories:

- *Fixed bugs* refer to those that have been confirmed by developers and subsequently addressed with fixing patches.
- *Confirmed bugs* refer to the bugs that have been acknowledged by developers, but have not been fixed.
- *Unconfirmed bugs* refer to the bugs that are identified manually according to the usage documentation and are awaiting developer confirmation.
- *Duplicate bugs* refer to bugs confirmed to have the same cause as previously confirmed bugs.

We consider 34 of them as previously unknown, unique bugs, 30 of which have been confirmed or fixed by the developers. Most of them affected PostGIS, as its developers quickly responded and addressed many of the issues we reported. We detected 2 bugs in SQL Server and reported them to the SQL Server community, but have not received any response; hence, we ceased testing it. One of the reported bugs was a duplicate, caused by the same root cause as a previously confirmed bug.

Table 3 shows the previously unknown and confirmed bugs, which are classified into logic and crash bugs. Out of the 30 bugs, the majority, 20 bugs were logic bugs that we aimed to find. We detected 9 logic bugs in GEOS and 7 SDBMS-specific logic bugs in PostGIS. Besides, Spatter detected 10 crash bugs. All of the crash bugs were fixed within one day. Logic bugs are more difficult to fix than crash bugs, because pinpointing the root cause is usually more difficult. At the same time, developers must ensure that patches do not affect other scenarios. For some bugs, PostGIS may need to propose and implement a new algorithm to address them, as indicated by developer feedback: *"There is a new topological predicate algorithm being worked on which solves this problem. It should be ready by mid-2024. "*

Table 2. Status of the reported bugs in SDBMSs. GEOS is a third-party library used by PostGIS and DuckDB Spatial. The bugs detected in GEOS are listed separately.

SDBMS	Fixed	Confirmed	Unconfirmed	Duplicate	Sum
GEOS	4	8	0	0	12
PostGIS	8	1	1	1	11
DuckDB Spatial	5	0	1	0	6
MySQL	1	3	0	0	4
SQL Server	0	0	2	0	2
Sum	18	12	4	1	35

Table 3. A classification of the confirmed and fixed bugs. Note that 2 fixed logic bugs are not listed here, because they were found in JTS, which is not an SDBMS.

	Logic Bugs		**Crash bugs**		
SDBMS	Fixed	Confirmed	Fixed	Confirmed	Sum
GEOS	1	8	3	0	12
PostGIS	6	1	2	0	9
MySQL	1	3	0	0	4
DuckDB Spatial	0	0	5	0	5
Sum	8	12	10	0	30

Impact of Spatter. Developer feedback is an important indicator of whether the testing work has a positive impact in practice. We received much positive feedback from PostGIS developers. They noticed our efforts and reached out to us via email: *"Congratulations on your thorough work in this area - it benefits all uses of PostGIS and GEOS!"* They emphasized the significant contribution of our approach in detecting logic bugs: *"I'm aware of some work around automated fuzzing tests for GEOS, but typically they just involve obscure input causing crashes, not simple test cases producing wrong answers."* In fixing one of the PostGIS bugs, a developer noted that a test case revealed an incorrect definition. They acknowledged, *"it has become clear to me that our definition of ST_DFullyWithin is probably "wrong", that is, it's not what people think they are getting when they call it."* From DuckDB Spatial, we received numerous appreciations from one of the core developers, saying, *"Thanks for reporting this issue!"* All the bugs were resolved and 4 of them were fixed within 24 hours. From MySQL, our reported bugs were confirmed by a testing developer within one day, one of which was marked as *serious*. Besides, a bug was fixed and included in the subsequent release version.

5.2 Illustrative Examples

In this section, we discuss selected bugs to illustrate why *AEI* can detect them, whereas differential testing cannot. We then present our results from manually analyzing all the bugs detected by *AEI* and determine how many of these bugs could also be identified by differential testing or TLP. Lastly, we identify common patterns that induce bugs to provide insights for implementing SDBMSs.

Selected bugs. We selected a few examples of logic bugs identified by Spatter to illustrate the effectiveness of *AEI* in bug detection, evidenced by the diversity of detected bugs. The variety in their root causes further underscores this diversity. For conciseness, we present only simplified test cases that highlight the underlying core issues, rather than the original test cases used to uncover these bugs.

Listing 3. MySQL calculates an incorrect relation after scaling coordinates by a factor of 10.

```
SET @g1='MULTILINESTRING((990 280,100 20))';
SET @g2='GEOMETRYCOLLECTION(MULTILINESTRING((990 280,    100 20)),POLYGON((360 6
  0,850 620,850 420,360 60)))';
SELECT ST_Crosses(ST_GeomFromText(@g1), ST_GeomFromText(@g2));
-- {1} 🐞 {0} ✔
```

Listing 4. MySQL identifies an incorrect relation after swapping the X and Y axes.

```
SET @g1 = ST_GeomFromText('POLYGON((614 445,30 26,80 30,614 445))');
SET @g2 = ST_GeomFromText('GEOMETRYCOLLECTION(POLYGON((614 445,30 26,80 30,614 445))
  ,POLYGON((190 1010,40 90,90 40,190 1010)))');
SELECT ST_Overlaps(@g2, @g1);
-- {0} ✔
SELECT ST_Overlaps(ST_SwapXY(@g2), ST_SwapXY(@g1));
-- {1} 🐞 {0} ✔
```

Incorrect result after scaling in MySQL. Listing 3 illustrates a logic bug related to an incorrect relation calculation. MySQL incorrectly determines that `@g1` crosses `@g2`, violating the condition that *"their intersection is not equal to either of the two given geometries"*. The results before and after scaling demonstrate this inconsistency. After identifying that the result containing an *EMPTY* element was incorrect, we reported it to the MySQL community.

The above bug is a logic bug that is difficult to detect by differential testing, as it is concealed within expected discrepancies. The definition of the function `ST_Crosses` varies among different SDBMSs, leading to expected discrepancies.

Incorrect result after swapping axes in MySQL. Listing 4 shows a confirmed logic bug in MySQL. The Function `ST_Overlaps(g1, g2)` returns `1` if `g1` and `g2` intersect and their intersection results in geometry of the same dimension but not equal to either `g1` or `g2`; otherwise, it returns `0`. In this case, since the intersection of `g1` and `g2` is equal to `g1`, the expected result of `ST_Overlaps(g1, g2)` should be `0`.

This logic bug cannot be detected by differential testing. PostGIS and DuckDB Spatial consider `g2` invalid, because two elements of `g2` intersect, resulting in a self-intersection error. This serves as an example of how different data designs can limit the effectiveness of differential testing approaches.

EMPTY-related bugs in PostGIS. Listing 5 shows a fixed logic bug in PostGIS. `ST_Distance` calculates the distance between the given two geometries `g1` and `g2`. The function returns the minimum distance if the given geometries are *MULTI geometries*. Thus, the expected result is 2. However, PostGIS returns 3 incorrectly. The developer repaired the incorrect recursive logic after we reported it. This bug was found by *canonicalization*.

This example shows a logic bug that can be missed by comparing SDBMSs with and without supporting `EMPTY` elements.

Incorrect strategy in computing boundary of the MIXED geometry in GEOS. Listing 6 demonstrates a logic bug caused by undefined behavior when evaluating *MIXED geometry* boundaries. According to the definition of `ST_Within`, a geometry is within another only if their interiors share at least one point. Both interiors of `g1` and `g2` contain `POINT(0 0)`, thus, the expected result is that `g1` is within `g2`. However, PostGIS incorrectly judges the relation and returns false. This was detected by canonicalization. Replacing g2 with `GEOMETRYCOLLECTION(LINESTRING(0 0,1 0),POINT(0 0))` yields an inconsistent result. A PostGIS developer in PostGIS confirmed it as an upstream

Listing 5. PostGIS calculates a distance incorrectly.

```
SELECT ST_Distance('MULTIPOINT((1 0),(0 0))'::geometry,        'MULTIPOINT((-2 0),
  EMPTY)'::geometry);
-- {3} 🐞  {2} ✔
SELECT ST_Distance('MULTIPOINT((1 0),(0 0))'::geometry,        'POINT(-2 0)'::
  geometry);
-- {2} ✔
```

Listing 6. PostGIS misjudges the relationship between g1 and g2.

```
SELECT ST_Within(g1,g2) FROM (SELECT 'POINT(0 0)'::geometry As g1, '
  GEOMETRYCOLLECTION(POINT(0 0),LINESTRING(0 0,1 0))'::geometry As g2);
-- {f} 🐞 {t} ✔
```

Listing 7. PostGIS misses one of the pairs.

```
CREATE table t (id int, geom geometry);
INSERT INTO t (id, geom) VALUES
    (1,'GEOMETRYCOLLECTION(MULTIPOINT((0 0),(3 1)))'::geometry),
    (2,'GEOMETRYCOLLECTION(MULTIPOINT((0 0),(3 1)))'::geometry),
    (3,'MULTIPOLYGON(((0 0,5 0,0 5,0 0)))'::geometry);
SELECT a1.id, a2.id FROM t As a1, t As a2 WHERE ST_Contains(a1.geom, a2.geom);
-- {1,1; 1,2; 2,1; 2,2; 3,1; 3,3} 🐞
-- {1,1; 1,2; 2,1; 2,2; 3,1; 3,2; 3,3} ✔
```

bug and reported it to the GEOS community. The GEOS developers' feedback indicated this bug was caused by a *"last-one-wins"* strategy, which dictates that a point serves as the boundary for a `GEOMETRYCOLLECTION` if it is the boundary of the last geometry. In this case, `POINT(0 0)`, the shared interior, is considered part of the boundary of `g2` rather than its interior. Developers considered adopting boundary-priority semantics to address it.

It is a logic bug that can be detected by differential testing, as PostGIS and MySQL output different results. However, when comparing the behavior of PostGIS and DuckDB Spatial, the bug would be missed, as they both return the same incorrect results.

A logic bug in prepared geometry of GEOS. Listing 7 demonstrates a logic bug where the ordered id pair `(3,2)` is missed in the results. `ST_Contains(g1, g2)` is defined to return true when `g1` contains `g2` if, and only if, all points of `g2` lie inside and the interiors of `g1` and `g2` share at least one point. Since the points in the second geometry lie within the third geometry, and `POINT(3 1)` is within the interior of the third geometry, the third geometry should contain the second geometry. It is noticed that the first and second geometry are the same. However, PostGIS reports the pair of (3,1), instead of (3,1;3,2). A PostGIS developer identified it as an upstream bug from GEOS. Feedback from GEOS developers indicated that the issue was within *"prepared geometry"*, a component designed to speed up various topological relationship functions. The bug was fixed within one day of confirmation. One of the developers believes this inconsistency is widespread in PostGIS, stating, *"as a general proposition, every prepared variant should return the same as the non-prepared variant; I imagine there's a lot of possible issues hiding in that proposition."*

It is a logic bug that can be detected by differential testing, as both MySQL and DuckDB Spatial produce the correct results.

Listing 8. PostGIS engine misses 2 pairs.

```
CREATE TABLE t AS SELECT 1 AS id, 'POINT EMPTY'::geometry AS geom;
CREATE INDEX idx ON t USING GIST (geom);
SET enable_seqscan = false;
SELECT COUNT(*) FROM t WHERE geom ~= 'POINT EMPTY'::geometry;
-- {0} 🐞 {1} ✔
```

Listing 9. A RANGE functionality fails in PostGIS.

```
SELECT ST_DFullyWithin('LINESTRING(0 0,0 1,1 0,0 0)'::geometry,'POLYGON((0 0,0 1,1
  0,0 0))'::geometry,100);
-- {f} 🐞 {t} ✔
```

A logic bug in the PostGIS engine. Listing 8 shows a fixed logic bug related to a GIST index. The correct result should be 1, but PostGIS incorrectly returns 0. We detected 7 bugs in the PostGIS engine, highlighting the diversity of detected bugs.

A bug in the RANGE functionality of PostGIS. Listing 9 shows a bug in the functionality of `ST_DFullyWithin` in PostGIS. The function returns true if the geometries are entirely within the specified distance of one another. The geometry a1 is fully within each point of a2 in the distance of 100, thus, it should return true. A PostGIS developer pointed out, *"It has become clear to me that our definition of* `ST_DFullyWithin` *is probably 'wrong', that is, it's not what people think they are getting when they call it."*

Patterns of inducing cases. We also identified common bug-inducing patterns, classifying them under *EMPTY* and *MIXED geometry* categories, providing a taxonomy of bugs based on trigger case patterns. Among all 20 logic bugs, 6 can be triggered by test cases containing *EMPTY* elements or geometries. These types of bugs are typically fixed within one day because pinpointing the root cause (*i.e.*, empty processor) is usually easier than with other patterns. Besides, 13 bugs are related to the *MIXED geometry*, caused by various factors. Excluding 4 bugs whose causes were not detailed in the feedback of MySQL developers, we summarize the causes as follows. 4 logic bugs stem from errors in processing *EMPTY* elements. Another 3 logic bugs are due to dimension processors incorrectly identifying the dimensions of *MIXED geometry*. Additionally, we detected 2 logic bugs in the *"prepared geometry"*, a component for optimization in PostGIS. Moreover, the root cause of Listing 6 is associated with the boundary processor of *MIXED geometry*.

5.3 Comparison to the State of the Art

To the best of our knowledge, Spatter is the first general testing approach and tool that aims to detect logic bugs for SDBMSs. Despite this, we undertook additional manual analyses of the logic bugs detected by *AEI*, to ascertain if they could have been identified by previous methodologies. For this purpose, we employed differential testing approaches alongside TLP [36], a state-of-the-art methodology for testing relational DBMSs. In comparing different SDBMSs, we chose PostGIS and DuckDB Spatial to illustrate a comparison between similar systems, and PostGIS with MySQL to demonstrate a comparison between more distinct systems. Aside from comparing different SDBMSs, we also conducted differential testing by toggling an index on and off, indicated as an approach *Index*. All 20 confirmed logic bugs detected by *AEI* were manually analyzed to determine their detectability by previous methodologies. When comparing different SDBMSs, we assessed the potential for bug detection by examining two factors in each report: (1) the functions or data in the bug-inducing case are applicable for the compared SDBMSs; (2) the bug-inducing case

Table 4. Logic bugs detection comparison. *P. vs. M.* compares the results of PostGIS and MySQL; *P. vs. D.* compares the outcomes of PostGIS and DuckDB Spatial. *Index* compares the outcomes of SDBMSs with and without index. *TLP* is a state-of-the-art methodology for testing relational DBMSs [36].

	AEI	***P. vs. M.***	***P. vs. D.***	***Index***	***TLP***
GEOS	8	3	1	0	0
PostGIS	8	0	0	1	1
MySQL	4	1	0	1	0
Sum	20	4	1	2	1

produced different results in the compared SDBMSs. Regarding *Index*, we analyzed the impact of index presence on each bug-inducing case within a specific SDBMS. For TLP, we decomposed each bug-inducing query into three partitioning queries and examined the results; unexpected outcomes signified TLP's potential for bug detection.

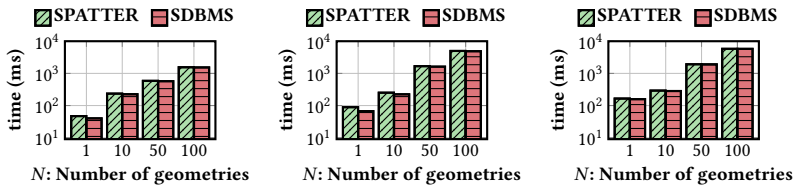
Table 4 shows the results of logic bug detection by different methods. Of the 20 confirmed or fixed logic bugs, 14 were overlooked by all other methods. 4 logic bugs could be detected by comparing PostGIS and MySQL; however, such differential testing suffers from false alarms, due to differing function definitions. All logic bugs were missed when comparing PostGIS and DuckDB Spatial, which is expected, given their similarity. Two index-related bugs could be found, both theoretically detectable by the *Index* method. However, applying the *Index* method heavily depends on the test case design, specifically designed to make frequent use of the index, thereby enabling more efficient application of this method. TLP detected one index-related bug, but missed the other, as expected, since TLP is designed for relational DBMS and lacks awareness of spatial relationships.

5.4 Efficiency of Spatter

Run time distribution. We evaluated whether the bottleneck of Spatter lies in the execution time of the SDBMS or in our test case generation. We varied a set of parameters, N, which controls the number of geometries in each run, to assist users in selecting an appropriate quantity of geometries for use in Spatter.

Methodology. We varied geometry generation per run by an argument, setting N—the number of geometries per group—at 1, 10, 50, and 100 as configuration parameters. We configured Spatter to generate 100 random queries in each run. Each experiment was repeated 10 times to accommodate potential performance deviations. Subsequently, we recorded the execution time of statements in the targeted SDBMS and the total time of Spatter including the run time of the SDBMS.

Results. Figure 7 shows the average time spent in each configuration. Note that the execution time of Spatter includes also the time spent within the SDBMS. The proportion of statement execution time exceeds those of Spatter by 90% when N is greater than 10 in PostGIS and MySQL. In all configurations for DuckDB Spatial, the execution time of statements accounted for more than 90%. Besides, in all the SDBMSs, as N increases, the total runtime of Spatter also increases. This trend is particularly notable in DuckDB Spatial and MySQL. When N is 100, the runtime is 20 times longer than when N is 10. The results indicate that the execution time spent in the SDBMS dominates the overall execution costs. This observation has been previously noted in testing approaches for relational DBMSs [37], but it had not been systematically evaluated before. Moreover, the number of geometries impacts the performance of Spatter indirectly, by increasing the execution time within the SDBMS, aiding users in choosing an appropriate quantity of geometries when testing.



(a) PostGIS (b) MySQL GIS (c) DuckDB Spatial

Fig. 7. Average time in Spatter and the SDBMSs across 10 runs.

Code coverage. We sought to evaluate the code coverage of Spatter, as it provides insights into the components of the SDBMSs are tested.

Methodology. We ran Spatter on PostGIS for over 24 hours, collecting line coverage data for both PostGIS and GEOS. As our testing target, we selected PostGIS 3.4.3 and GEOS 3.12, the versions we used when we began our testing campaign. The reason for selecting PostGIS as a representative is that we found most bugs in PostGIS and its third-party library GEOS. We collected coverage data after executing its unit tests, as this allowed us to determine whether Spatter had an additional contribution to coverage on top of unit tests. In the configuration of *"Unit Tests + Spatter"*, we ran Spatter for 24 hours after executing all PostGIS unit tests. It is noticed that the unit tests are from PostGIS, not GEOS, since GEOS's unit tests cover functions (*e.g.*, distance computation) that PostGIS does not call. During our testing campaign, PostGIS and GEOS developers incorporated our reported cases into their regression unit tests upon bug resolution. Therefore, we selected a specific commit before our testing campaign.

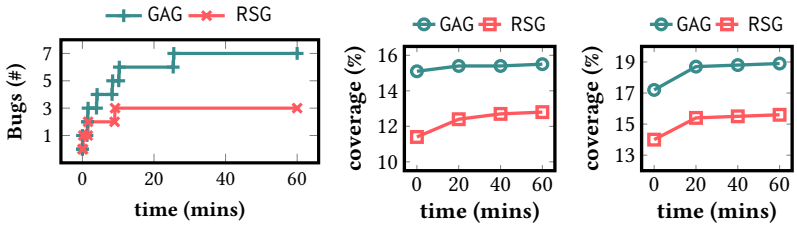
Consequences. Table 5 presents the coverage results in various settings. A line coverage of less than 20% in the PostGIS module might seem low, but this outcome was expected. The primary reason for low coverage is that spatial data engines encompass more components than just geometry relation processing. For example, in the PostGIS module, around 15% of the code pertains to I/O-related functions supporting various formats (*e.g.*, GeoJSON, a JSON format describing geometry or geography), and 8% of uncovered code is related to geography. In the GEOS module, the code coverage of around 20% was also expected. Approximately 18% of the code, which is related to geometry operations (around 20% of GEOS), remained uncovered, because we deliberately exercised caution in calling geometry operations within the *geometry-aware generator* to prevent precision issues. Spatter is designed to detect bugs in queries concerning topological relationships rather than spatial analysis or operation functions, resulting in low function coverage. However, coverage increased after running Spatter based on unit tests. Specifically, in PostGIS and GEOS, an additional 206 and 178 lines of code, respectively, were covered, indicating that Spatter provides supplementary coverage beyond that of unit tests. These results are expected, as most logic bugs tend to occur within a limited number of code lines [42].

Self-constructed baseline. We constructed our own baseline to evaluate the efficiency of the *random-shape strategy*, which, to the best of our knowledge, represents the first automated test case generation tool for SDBMSs. Our baseline generates geometry based solely on the *random-shape strategy*, providing a comparison point to assess the efficiency of the *geometry-aware generator*.

Methodology. We selected PostGIS version 3.4 as our evaluation target. The reason for selecting PostGIS was that it had the highest number of bug fixes, allowing us to automatically determine the number of unique bugs we found during a testing campaign. To identify unique bugs over time,

Table 5. Code coverage of the systems tested over 24 hours.

Approach	PostGIS		GEOS	
	Line	Function	Line	Function
Spatter	15.8%	13.9%	20.1%	18.8%
Unit Tests	79.5%	76.7%	54.8%	56.2%
Unit Tests + Spatter	79.9%	76.8%	55.2%	56.7%



(a) Unique Bugs (b) PostGIS (c) GEOS

Fig. 8. Ablation study of the Geometry-Aware Generator (*GAG*). *RSG* denotes a generator that employs only the *random-shape strategy*. Sub-figure (a) shows the number of unique bugs in PostGIS identified under different configurations over one hour. Sub-figure (b) and (c) illustrate the corresponding line coverage of PostGIS and GEOS over time.

we performed the following steps. First, we ran Spatter on specific older versions for one hour to collect bug-inducing cases and their trigger timestamps. Second, for each bug-inducing test case, we determined whether the bug was fixed by updating PostGIS and GEOS to their latest versions, respectively, and checking the consistency of the test case results. If a test case was identified as fixed, we concurrently identified the module from which the bug originated. Then, we conducted a binary search for the fix commits in the commits of PostGIS or GEOS. Finally, based on the commit and generation time of each fixed case, we determined the earliest detection time for each fixed bug during the experiment. This approach is a best-effort technique; for instance, a single fix commit might address multiple bugs. We configured each test to run for an hour. Meanwhile, we collected coverage data for PostGIS and GEOS over time.

Results. Our testing resulted in 2,366 and 9,913 cases potentially triggering bugs under configurations with and without the *derivative strategy*, respectively. As expected, most of them were duplicated. We applied the deduplication technique described in our methodology to address this issue. As Figure 8 shows, the generator with our proposed strategy, the *derivative strategy*, significantly outperformed the generator only using the *random-shape strategy*. Within one hour, the *geometry-aware generator* detected a higher number of unique bugs, since the *derivative strategy*, which generates more complex topological relationships, rendered the query results more significant. Furthermore, the *geometry-aware generator* achieved higher coverage in both PostGIS and GEOS, which aligns with our expectations, given that the *derivative strategy* exploits spatial functions inherent in PostGIS. The findings underscore that the *derivative strategy*, which creates new geometries based on the existing geometries, enhances testing efficiency.

6 Related Work

Testing DBMSs. Recently, various techniques have been proposed to detect bugs in DBMSs. For relational DBMSs, researchers have proposed methods to identify logic bugs [1, 8, 20, 36–38], transactional bugs [4, 8, 17, 19, 22, 43], and performance issues [2, 20, 25]. A notable example is SQLancer, which integrates several novel approaches to identify a range of bug types. TLP [36], starting from a given original query, derives multiple, more complex queries, each of which computes a partition of the result. CERT utilizes the property of cardinality estimation to efficiently detect performance bugs. For graph DBMSs, established methodologies such as *differential testing* [46] and TLP [21] have been utilized to uncover logic bugs. Besides, methods leveraging graph properties have been proposed to specifically address logic bugs in graph query processing [18, 26, 48]. However, these testing techniques largely depend on the properties of the specific type of DBMS under test and are not directly aimed at evaluating the core functionality of SDBMSs, particularly geometry processing. To the best of our knowledge, Spatter represents the first approach specifically designed for SDBMSs, efficiently testing both the functionalities shared with other DBMS types and those unique to SDBMSs.

SDBMSs. Various approaches have focused on studying and improving the performance of SDBMSs. A multitude of spatial indexes have been proposed, including the R-tree [16], R*-tree [3], and machine learning-based indexes [14, 24]. In addition, various optimizations for spatial joins [13, 27, 39, 40] have been proposed. Other advancements include learned components within SDBMSs [30], as well as end-to-end systems [44]. For a systematic study of modern SDBMSs, we defer interested readers to a study by Pandey et al. on modern SDBMSs and their performance on real-world datasets [29].

7 Discussion

The reason why AEI *detected bugs.* We analyzed why *AEI* detects bugs by examining their root causes. Given that we check for a high-level property, *AEI* can detect a diverse range of bugs. Commonly, the reason is that the original input exercises a different path in the program as compared to the transformed follow-up database. For example, the bug in Listing 1 is located in a function that computes the direction of a point p relative to a vector v. There was a loss of precision in the normalization of vertices (*i.e.*, displacement to the origin). The statements in Listing 2 fail to trigger this bug, because a point in v is at the origin, leading to no displacement calculations. Listing 5 is another example that illustrates how *canonicalization* detected bugs. It was caused by incorrect logic when recursively processing MULTI geometry. Line 3 in Listing 5 fails to trigger this bug, because the recursive logic is not needed to handle the second geometry. Overall, *AEI* can effectively detect bugs, because the inputs before and after transformation exercise different paths.

Supported SDBMS-specific functionality. We support various complex kinds of SDBMS-specific functionality. For JOIN functionality, all of our test cases use JOIN to combine spatial data, given that spatial join is one of the most important functionalities in SDBMSs. For the RANGE functionality, SDBMSs support functions like `ST_Within`, `ST_DWithin`, and `ST_DFullyWithin` for spatial range queries. We have tailored functionalities based on *AEI* to test these functions and have indeed found significant bugs (*e.g.*, see Listing 9). In general, *AEI* can effectively test functionalities highly related to the geometry in SDBMSs, such as topological relationship queries.

Applicability to other kinds of database systems. AEI is a general approach and might also be applicable to other kinds of database systems. For example, KNN algorithms allow nearest-neighbour searching and are commonly supported not only by geospatial database systems, but also by other systems, such as vector database systems. It is used for classification, dimensionality reduction, and

content recommendation. While currently not supported in our implementation, testing for KNN algorithms using *AEI* could be implemented as long as no shearing is applied to the geometric objects by performing the following steps: (1) creating a database SDB1 by the *geometry-aware generator*; (2) canonicalizing each geometry in SDB1 to its equivalent representation and then applying an affine transformation (rotate, translate, or scale) to construct a new geometry, resulting in SDB2; (3) checking that the KNN results retrieved from SDB1 and SDB2 should be equal, otherwise, a logic bug is detected, since rotating, translating, and scaling preserve relative distance. Shearing cannot be applied, as it does not preserve the relative-distance property, thus the results of KNN queries before and after shearing could be inconsistent.

Limitations of AEI. While *AEI* is effective in identifying certain types of logic bugs, it does not cover all possible scenarios. First, *AEI* is built on affine transformations that preserve geometric properties, which do not directly apply to geography types, since they represent curved objects. Second, *AEI* is not designed to comprehensively test components tangential to query processing, such as reading and conversion of files (*e.g.*, implemented via the GDAL library). For instance, we detected a bug in GDAL by differential testing of the SDBMSs instead of *AEI*. DuckDB Spatial was expected to accept the GeoJSON {"type": "Polygon","coordinates": []} as a representation of `POLYGON EMPTY`, but, unexpectedly, the result was `NULL`. Furthermore, *AEI* is inherently unsuitable for applications involving non-linear transformations or non-affine geometric operations. For example, `ST_Buffer` cannot be used after *affine transformations* in *AEI* construction, as it fails to preserve topological relationships.

Future work. There are several future directions for this work. First, *AEI* could be applied to test other geometry libraries. We found that 12 out of 21 confirmed bugs in PostGIS were in the geometry library. Given that the underlying geometry libraries are prone to bugs, a future direction could be to apply *AEI* directly to these libraries. It would also help to further validate the robustness and effectiveness of Spatter in a different geospatial ecosystem. Besides, symbolic execution could enhance *AEI* by exploring more execution paths. Specifically, symbolic execution can generate test cases that explore different execution paths, with *AEI* providing expected results, thereby replacing the *geometry-aware generator*. Furthermore, the core idea of *AEI* that transforms a database while preserving essential properties can be generalized to testing other DBMSs, such as GDBMSs. For example, after shuffling the identities between all vertices in a graph database, the results of pattern matching should be still unchanged.

8 Conclusion

Logic bugs in SDBMSs related to spatial patterns are a severe problem. In this work, we proposed Spatter, which is based on a new methodology called *Affine Equivalent Inputs* (*AEI*) and a *geometry-aware generator* to effectively and efficiently detect logic bugs in SDBMSs. Our evaluation of Spatter on four widely-used SDBMSs found 34 previously unknown, and unique bugs. 30 of them have been confirmed, and 18 have already been fixed. Further bug analysis shows that 14 bugs are overlooked by the previous methodologies. Additionally, the *derivative strategy* in the generating process improves testing efficiency by finding more unique bugs and achieving higher coverage. We believe that Spatter, given its simplicity and generality, has a high chance of being integrated into the toolbox of SDBMS developers.

Acknowledgments

This research was supported by a Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant, and by 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 Amazon. This research was partly conducted while Wenjing Deng and Qiuyang Mang visited the National University of Singapore.

References

[1] Ba, J., and Rigger, M. Testing database engines via query plan guidance. In *Proceedings of the 45th International Conference on Software Engineering* (2023), ICSE '23, IEEE Press, p. 2060–2071.

[2] Ba, J., and Rigger, M. Cert: Finding performance issues in database systems through the lens of cardinality estimation. In *The 46th International Conference on Software Engineering (ICSE'24)* (Apr. 2024).

[3] Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. The r*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1990), SIGMOD '90, Association for Computing Machinery, p. 322–331.

[4] Biswas, R., and Enea, C. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang. 3*, OOPSLA (oct 2019).

[5] Byer, O., Lazebnik, F., and Smeltzer, D. L. *Methods for Euclidean Geometry*, vol. 37. Classroom Resource Materials, 2010.

[6] Clementini, E., Di Felice, P., and van Oosterom, P. A small set of formal topological relationships suitable for end-user interaction. In *Advances in Spatial Databases* (Berlin, Heidelberg, 1993), D. Abel and B. Chin Ooi, Eds., Springer Berlin Heidelberg, pp. 277–295.

[7] Clementini, E., Sharma, J., and Egenhofer, M. J. Modelling topological spatial relations: Strategies for query processing. *Computers & Graphics 18*, 6 (1994), 815–822.

[8] Cui, Z., Dou, W., Dai, Q., Song, J., Wang, W., Wei, J., and Ye, D. Differentially testing database transactions for fun and profit. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2023), ASE '22, Association for Computing Machinery.

[9] Egenhofer, M., and Herring, J. Categorizing binary topological relations between regions, lines and points in geographic databases, the 9-intersection: Formalism and its use for natural language spatial predicates. *Santa Barbara CA National Center for Geographic Information and Analysis Technical Report 94* (01 1990), 1–28.

[10] Egenhofer, M. J., Frank, A. U., and Jackson, J. P. A topological data model for spatial databases. In *Design and Implementation of Large Spatial Databases* (Berlin, Heidelberg, 1990), A. P. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang, Eds., Springer Berlin Heidelberg, pp. 271–286.

[11] Fan, Z., Chen, R., and Chen, X. SpatialDB: a database for spatially resolved transcriptomes. *Nucleic Acids Research 48*, D1 (11 2019), D233–D237.

[12] Felice, P. D., and Clementini, E. *Topological Relationships*. Springer US, Boston, MA, 2009, pp. 3140–3143.

[13] Georgiadis, T., and Mamoulis, N. Raster intervals: An approximation technique for polygon intersection joins. *Proc. ACM Manag. Data 1*, 1 (may 2023).

[14] Gu, T., Feng, K., Cong, G., Long, C., Wang, Z., and Wang, S. The rlr-tree: A reinforcement learning based r-tree for spatial data. *Proc. ACM Manag. Data 1*, 1 (may 2023).

[15] Güting, R. H. An introduction to spatial database systems. *the VLDB Journal 3* (1994), 357–399.

[16] Guttman, A. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (1984), pp. 47–57.

[17] Huang, K., Liu, S., Chen, Z., Wei, H., Basin, D., Li, H., and Pan, A. Efficient black-box checking of snapshot isolation in databases. *Proc. VLDB Endow. 16*, 6 (feb 2023), 1264–1276.

[18] Jiang, Y., Liu, J., Ba, J., Yap, R. H. C., Liang, Z., and Rigger, M. Detecting logic bugs in graph database management systems via injective and surjective graph query transformation. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)* (2023), IEEE Computer Society, pp. 531–542.

[19] Jiang, Z.-M., Liu, S., Rigger, M., and Su, Z. Detecting transactional bugs in database engines via Graph-Based oracle construction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (Boston, MA, July 2023), USENIX Association, pp. 397–417.

[20] Jung, J., Hu, H., Arulraj, J., Kim, T., and Kang, W. Apollo: Automatic detection and diagnosis of performance regressions in database systems. *Proc. VLDB Endow. 13*, 1 (sep 2019), 57–70.

[21] Kamm, M., Rigger, M., Zhang, C., and Su, Z. Testing graph database engines via query partitioning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2023), ISSTA 2023, Association for Computing Machinery, p. 140–149.

[22] Kingsbury, K., and Alvaro, P. Elle: inferring isolation anomalies from experimental observations. *Proc. VLDB Endow. 14*, 3 (nov 2020), 268–280.

[23] Kriegel, H.-P., Müller, A., Pötke, M., and Seidl, T. Spatial data management for computer aided design. *SIGMOD Rec. 30*, 2 (may 2001), 614.

[24] Li, P., Lu, H., Zheng, Q., Yang, L., and Pan, G. Lisa: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2020), SIGMOD '20, Association for Computing Machinery, p. 2119–2133.

[25] Liu, X., Zhou, Q., Arulraj, J., and Orso, A. Automatic detection of performance bugs in database systems using equivalent queries. In *Proceedings of the 44th International Conference on Software Engineering* (New York, NY, USA, 2022), ICSE '22, Association for Computing Machinery, p. 225–236.

[26] Mang, Q., Fang, A., Yu, B., Chen, H., and He, P. Testing graph database systems via equivalent query rewriting. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA, 2024), ICSE '24, Association for Computing Machinery.

[27] Nobari, S., Tauheed, F., Heinis, T., Karras, P., Bressan, S., and Ailamaki, A. Touch: in-memory spatial join by hierarchical data-oriented partitioning. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, Association for Computing Machinery, p. 701–712.

[28] Open Geospatial Consortium. OpenGIS Web Map Service (WMS) Implementation Specification. OGC 01-068r3, 2006. Version 1.3.0.

[29] Pandey, V., Kipf, A., Neumann, T., and Kemper, A. How good are modern spatial analytics systems? *Proc. VLDB Endow. 11*, 11 (jul 2018), 1661–1673.

[30] Pandey, V., van Renen, A., Kipf, A., Sabek, I., Ding, J., and Kemper, A. The case for learned spatial indexes. *ArXiv abs/2008.10349* (2020).

[31] Piórkowski, A., et al. Mysql spatial and postgis–implementations of spatial data standards. *Electronic journal of Polish agricultural universities 14*, 1 (2011), 1–8.

[32] PostGIS. Postgis (actually liblwgeom) integration with oss-fuzz. https://lists.osgeo.org/pipermail/postgis-devel/2017-July/026216.html. Retrieved March 1, 2024.

[33] Raasveldt, M., and Mühleisen, H. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 1981–1984.

[34] Rigaux, P., Scholl, M., and Voisard, A. *Spatial databases: with application to GIS*. Morgan Kaufmann, 2002.

[35] Rigger, M., and Su, Z. 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* (New York, NY, USA, 2020), ESEC/FSE 2020, Association for Computing Machinery, p. 1140–1152.

[36] Rigger, M., and Su, Z. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang. 4*, OOPSLA (nov 2020).

[37] Rigger, M., and Su, Z. Testing database engines via pivoted query synthesis. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation* (USA, 2020), OSDI'20, USENIX Association.

[38] Song, J., Dou, W., Cui, Z., Dai, Q., Wang, W., Wei, J., Zhong, H., and Huang, T. Testing database systems via differential query execution. In *Proceedings of the 45th International Conference on Software Engineering* (2023), ICSE '23, IEEE Press, p. 2072–2084.

[39] Tauheed, F., Heinis, T., and Ailamaki, A. Thermal-join: A scalable spatial join for dynamic workloads. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, Association for Computing Machinery, p. 939–950.

[40] Tsitsigkos, D., Bouros, P., Mamoulis, N., and Terrovitis, M. Parallel in-memory evaluation of spatial joins. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (New York, NY, USA, 2019), SIGSPATIAL '19, Association for Computing Machinery, p. 516–519.

[41] Wiki. DE-9IM. https://en.wikipedia.org/wiki/DE-9IM. Retrieved January 14, 2024.

[42] Winterer, D., Zhang, C., and Su, Z. On the unusual effectiveness of type-aware operator mutations for testing smt solvers. *Proc. ACM Program. Lang. 4*, OOPSLA (nov 2020).

[43] Xia, Y., Yu, X., Butrovich, M., Pavlo, A., and Devadas, S. Litmus: Towards a practical database management system with verifiable acid properties and transaction correctness. In *Proceedings of the 2022 International Conference on Management of Data* (New York, NY, USA, 2022), SIGMOD '22, Association for Computing Machinery, p. 1478–1492.

[44] Xie, D., Li, F., Yao, B., Li, G., Zhou, L., and Guo, M. Simba: Efficient in-memory spatial analytics. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, Association for Computing Machinery, p. 1071–1085.

[45] Zeller, A., and Hildebrandt, R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering 28*, 2 (2002), 183–200.

[46] Zheng, Y., Dou, W., Wang, Y., Qin, Z., Tang, L., Gao, Y., Wang, D., Wang, W., and Wei, J. Finding bugs in gremlin-based graph database systems via randomized differential testing. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2022), ISSTA 2022, Association for Computing

Machinery, p. 302–313.
[47] Zheng, Y., Xie, X., and Ma, W.-Y. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data Eng. Bull. 33* (2010), 32–39.
[48] Zhuang, Z., Li, P., Ma, P., Meng, W., and Wang, S. Testing graph database systems via graph-aware metamorphic relations. *Proc. VLDB Endow. 17*, 4 (mar 2024), 836–848.

Received April 2024; revised July 2024; accepted August 2024