Ternary Logic Partitioning: Detecting Logic Bugs in Database Management Systems

MANUEL RIGGER, Department of Computer Science, ETH Zurich, Switzerland
ZHENDONG SU, Department of Computer Science, ETH Zurich, Switzerland

Logic bugs in Database Management Systems (DBMS) are bugs that cause an incorrect result for a given query (e.g., by omitting a row that should be fetched). These bugs are critical, since they are likely to go unnoticed by users. We propose *Query Partitioning*, a general and effective approach for finding logic bugs in DBMS. The core idea of Query Partitioning is to, starting from a given original query, derive multiple, more complex queries (called *partitioning queries*), each of which computes a *partition* of the result. The individual partitions are then composed to compute a result set that must be equivalent to the original query's result set. A bug in the DBMS is detected when these result sets differ. Our intuition is that due to the increased complexity, the partitioning queries are more likely to stress the DBMS and trigger a logic bug than the original query. As a concrete instance of a partitioning strategy, we propose Ternary Logic Partitioning (TLP), which is based on the observation that a boolean predicate `p` can either evaluate to `TRUE`, `FALSE`, or `NULL`. Accordingly, a query can be decomposed to three partitioning queries, each of which computes its result on rows or intermediate results for which `p`, `NOT p`, and `p IS NULL` hold. This technique is versatile, and can be used to test `WHERE`, `GROUP BY`, as well as `HAVING` clauses, aggregate functions, and `DISTINCT` queries. As part of an extensive testing campaign, we found 175 bugs in widely-used DBMS such as MySQL, TiDB, SQLite, and CockroachDB, 123 of which have been fixed. Notably, 77 of these were logic bugs, while the remaining error and crash bugs. We expect that the effectiveness and wide applicability of Query Partitioning will lead to its broad adoption in practice, and the formulation of additional partitioning strategies.

ACM Reference Format:
Manuel Rigger and Zhendong Su. 2020. Ternary Logic Partitioning: Detecting Logic Bugs in Database Management Systems. 1, 1 (May 2020), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Database Management Systems (DBMS) are used ubiquitously. Most DBMS allow inserting, deleting, modifying, and querying data from a database using the *Structured Query Language (SQL)*. As with other software, DBMS can be affected by various kinds of bugs. In this work, we consider *logic bugs*, which we define as bugs that cause the DBMS to fetch an incorrect result set for a query. For example, for a given query, a DBMS might mistakenly omit a record from the result set, fetch a record that should not be in the result set, or compute an incorrect result for a function or operator. Such bugs are difficult to detect by users and might go unnoticed, especially considering the scale of many databases.

To tackle logic bugs in DBMS, we propose a general and effective technique to which we refer to as *Query Partitioning*. The core idea of Query Partitioning is, based on a given query Q with a result set $RS(Q)$, to derive n queries $Q'_0 \dots Q'_{n-1}$, each of which computes a partial result $RS(Q'_i)$. The n partial results can then be composed using a predefined, n-ary *composition operator* $\diamond$ to obtain a result set $RS(Q'_0) \diamond RS(Q'_1) \diamond \dots \diamond RS(Q'_{n-1})$. For simplicity, we denote the composed partial results as $RS(Q')$. The original query's result set and the composed partitions must be equal, that is, $RS(Q') = RS(Q)$. Bugs in the DBMS can then be detected by checking whether the equality indeed

Authors' addresses: Manuel Rigger, manuel.rigger@inf.ethz.ch, Department of Computer Science, ETH Zurich, Zurich, Switzerland; Zhendong Su, zhendong.su@inf.ethz.ch, Department of Computer Science, ETH Zurich, Zurich, Switzerland.

holds. While a number of partitioning strategies are imaginable, it is crucial to select one that stresses the DBMS and its query optimizer in different ways [Giakoumakis and Galindo-Legaria 2008], either between Q''s partitioning queries or between Q' and Q, so that an inconsistent result set might be observed.

As part of this work, we propose *Ternary Logic Partitioning (TLP)*, which can effectively test the correct implementation and optimization of **WHERE** clauses, **GROUP BY** clauses, **HAVING** clauses, aggregate functions, and **DISTINCT** clauses. SQL is based on a ternary boolean logic, which means that a predicate ϕ can either evaluate to **TRUE**, **FALSE** or **NULL**. The predicate can be interpreted as a *piecewise-defined* total function p, with the current row r as an argument:

$$p(r) = \begin{cases} \text{TRUE} & \text{if } \phi \\ \text{FALSE} & \text{if } \neg\phi \\ \text{NULL} & \text{if } \phi \text{ IS NULL} \end{cases}$$

Consider a random row r from $RS(Q)$. Irrespective of which predicate we might choose (or randomly generate), we know that exactly one of the conditions of the piecewise function p must hold. Based on this insight, we can partition any Q by deriving three queries that filter records based on whether p holds, $\neg p$ holds, or whether p is **NULL**, while guaranteeing that the combined result comprises all rows of the original query. If used to test a **WHERE** clause, the individual subqueries can be aggregated using a union operator.

Consider Listing 1, which demonstrates an unknown bug that we reported for MySQL version 8.0.19 and which was fixed for version 8.0.21. Query ① computes an incorrect result set, and demonstrates the underlying bug. One record consisting of the rows in `t0` and `t1` should be fetched, since `0` and `-0` represent the same number, so the comparison should evaluate to **TRUE**. We found this bug based on the original query Ⓞ and the partitioning queries Ⓟ. Ⓞ lacks a **WHERE** clause and thus fetches the cross product of all values in `t0` and `t1`; since both tables contain only a single record, only a single record is fetched. Ⓟ consists of three partitioning queries that are connected by the **UNION ALL** keyword, which combines the queries' result sets. We derived these queries by generating a random predicate `t0.c0 = t1.c0` for the **WHERE** clause, and then creating the two other variants with the negated predicate and `IS` **NULL** predicate. Thus, Ⓟ's result set is expected to be the same as the one for query Ⓞ. However, since the query with the predicate `t0.c0 = t1.c0` was processed incorrectly, and resulted in the omission of the row, we detected this bug. Based on Ⓞ and Ⓟ, we manually created the test case ① to report the bug.

Listing 1. A logic bug in MySQL caused a predicate `0=-0` to incorrectly evaluate to **FALSE**. The check symbol denotes the expected, correct result, while the bug symbol denotes the actual, incorrect result.

```
CREATE TABLE t0(c0 INT);
CREATE TABLE t1(c0 DOUBLE);
INSERT INTO t0 VALUES(0);
INSERT INTO t1 VALUES('-0');
① SELECT * FROM t0, t1 WHERE t0.c0 = t1.c0; -- {}🐞
Ⓞ SELECT * FROM t0, t1; -- {0, -0}✔
Ⓟ SELECT * FROM t0, t1 WHERE t0.c0 = t1.c0
        UNION ALL SELECT * FROM t0, t1 WHERE NOT(t0.c0 = t1.c0)
        UNION ALL SELECT * FROM t0, t1 WHERE (t0.c0 = t1.c0) IS NULL; -- {}🐞
```

Query Partitioning addresses fundamental limitations in existing approaches. *Pivoted Query Synthesis (PQS)* detects logic bugs by checking whether a randomly-selected *pivot row* is fetched correctly [Rigger and Su 2020b]. To construct a query that fetches the row, PQS relies on an implementation of the DBMS SQL dialect's supported operators and functions. The technique has proven to be effective. However, unlike Query Partitioning, its implementation effort is high and requires detailed knowledge of the DBMS' operator and function semantics. *Non-optimizing Reference Engine Construction (NoREC)* detects bugs in queries that use a `WHERE` predicate by rewriting the query to disable the DBMS' optimizations and addresses PQS' high implementation effort [Rigger and Su 2020a]. A major limitation of both NoREC and PQS is that they are applicable primarily to test `WHERE` predicates; while they can partially be used to test other features—for example, PQS can test `DISTINCT`, but cannot detect duplicate rows—it would be unclear, for example, how these approaches could be extended to test aggregate queries.

We evaluated the effectiveness of Query Partitioning in a large-scale study on six widely-used DBMS. We found 175 true, previously unknown bugs in five of these systems, which demonstrates the effectiveness and generality of the proposed approach. Many of these were considered important by the developers of the DBMS, and 123 of these bugs have already been fixed. 77 bugs were logic bugs, while the remaining error and crash bugs. Furthermore, we compared our proposed approach with NoREC. Our results suggest that TLP can detect 17 bugs in features that are out-of-scope for NoREC, and 12 additional bugs related to `WHERE` clauses, to which both are applicable. Ultimately, Query Partitioning is complementary to PQS, and shares the same advantages and disadvantages as NoREC, both being metamorphic test oracles [Chen et al. 1998]. Due to the high effectiveness and low implementation effort, we believe that our approach might be widely adopted in practice. For reproducibility, and to facilitate the adoption of TLP, we provide our implementation at *https://github.com/sqlancer.*

Overall, this paper contributes the following:

- Query Partitioning, a general technique designed for finding logic bugs in DBMS that use SQL as a query language;
- Ternary Logic Partitioning (TLP), an instantiation of Query Partitioning based on the insight that a boolean predicate can be partitioned to evaluate to `TRUE`, `FALSE`, or `NULL`;
- Concrete TLP oracles to test queries using `WHERE`, `HAVING`, and `GROUP BY` clauses as well as aggregate functions and `DISTINCT` queries; and
- An extensive evaluation of Query Partitioning on six widely-used DBMS, in which the technique found 175 bugs, and a comparison with the state-of-the-art approach NoREC.

2 BACKGROUND

Relational DBMS. DBMS are based on a *data model*, which abstractly describes how data is organized. We primarily aim to test DBMS based on the *relational data model* proposed by Codd [1970], on which most widely-used databases, such as Oracle, Microsoft SQL, PostgreSQL, MySQL, and SQLite are based. Relational DBMS use a domain-specific language, Structured Query Language (SQL), for interaction. SQL's data model is based on bags (*i.e.*, multisets), where the same row can occur multiple times [Guagliardo and Libkin 2017]. This contrasts the original relational model, which is based on the concept of sets. Since a query's result is typically nevertheless referred to as a *result set*, we also use this term in this paper. In order to merge two bags, without removing duplicates, the multiset addition, denoted as $\uplus$, is used. To exclude duplicate elements, the union operator, denoted as $\cup$, is used. In SQL, the `UNION ALL` operator corresponds to $\uplus$, and `UNION`—without `ALL`—to $\cup$. Both operators are used in the composition operator of different TLP test oracles.

SQL. We assume basic familiarity with SQL, and thus provide only a minimal overview of it. In our work, we concentrate on the `SELECT` statement, which allows querying data from a database. SQL provides various ways of filtering, grouping, and aggregating data. A `WHERE` clause can be used to specify which rows should be fetched. It contains a boolean *predicate*, which can evaluate to `TRUE`, `FALSE`, or `NULL`. A number of DBMS (*e.g.*, SQLite and MySQL) allow the usage of a predicate of any type in a `WHERE` clause, as they apply implicit conversions to convert values of other types to a boolean value. A `GROUP BY` clause can be used to aggregate rows. It specifies a number of expressions, based on which the DBMS groups rows for which the expression evaluate to the same value. They can be used in combination with `HAVING` clauses, which allow filtering rows after they are grouped. Similarly to `GROUP BY`, a query can contain a `DISTINCT` clause, to compute a result set rather than a bag (*i.e.*, the `DISTINCT` removes all duplicate rows). Aggregate functions compute values over multiple rows. They can either be used to aggregate the final result set, or in a `HAVING` clause as part of a predicate. SQL is a feature-rich language and provides a number of additional features (*e.g.*, window functions and transactions). While our core idea could be generalized to some additional features, we consider them less important and out of scope.

Aggregate Functions. Various kinds of aggregate functions exist, such as `MIN()` and `MAX()` to compute minimum and maximum values for an input expression, `SUM()` to sum up input values, `COUNT()` to count the number of rows, and `AVG()` to compute the average. We base our testing ideas for aggregate functions on the optimization of aggregate functions by distributing their computations [Cohen 2006; Jesus et al. 2015; Yu et al. 2009]. Important properties for aggregate functions were defined [Jesus et al. 2015]. An aggregate function f is *self-decomposable* when a merge operator $\oplus$ exists so that, given two non-empty multi-sets X and Y, the following holds: $f(X \uplus Y) = f(X) \oplus f(Y)$. Many functions, including `MIN()`, `MAX()`, `SUM()`, and `COUNT()` are self-composable. For example, consider `SUM()`: $SUM(\{x\}) = x$ and $SUM(X \uplus Y) = SUM(X) + SUM(Y)$. An aggregate function f is *composable* if for some function f and a self-decomposable aggregate function h, it can be expressed as $f = g \circ h$. Every self-composable function is composable, by assigning g as the identity function. The `AVG()` function is composable when defining g as $g((s, c)) = s/c$ and h as follows: $h(\{x\}) = (x, 1)$ and $h(X \uplus Y) = h(X) + h(Y)$. That is, the `AVG()` function is computed by dividing the sum of values by the number of rows: $AVG(X \uplus Y) = (SUM(X) + SUM(Y))/(COUNT(X) + COUNT(Y))$.

Automatic Testing. We propose a novel automatic testing approach for DBMS. Two components are essential. First, an effective test case should stress significant portions of the system under test. To this end, a number of database generators have been proposed [Binnig et al. 2007b; Bruno and Chaudhuri 2005; Gray et al. 1994; Houkjær et al. 2006; Khalek et al. 2008; Neufeld et al. 1993], as well as a number of query generators [Bati et al. 2007; Bruno et al. 2006; Jung et al. 2019; Mishra et al. 2008; Poess and Stephens 2004; Seltenreich 2019; Vartak et al. 2010]. While these are important components of an overall testing approach, they are well understood, and thus not the main focus of this paper. We believe that any database generator and query generator that provides control over the format of the queries generated can be used to find bugs using Query Partitioning. In the implementation of our approach, we use SQLancer's database and query generation mechanism [Rigger and Su 2020a,b]. To create a database, SQLancer heuristically selects applicable options such as `CREATE TABLE` to create a table, `CREATE INDEX` to create indexes, and `INSERT` to insert data into relations. To create a query, SQLancer heuristically selects applicable operators and column names (which are leaf nodes in the AST). Second, an effective *test oracle* needs to determine whether the generated test case's result is correct. A specific class of test oracles are *metamorphic* ones, which can derive a test case and its expected result based on an input and output of a system [Chen et al. 1998]. While the implementation effort for such oracles is often low, they cannot provide a ground truth (*i.e.*, since the output based on which the new test case is generated might be incorrect). The main focus of this paper is metamorphic test oracles, which are based on the general idea of Query Partitioning.

Pivoted Query Synthesis. Pivoted Query Synthesis (PQS) was recently proposed by [Rigger and Su 2020b] to find logic bugs in DBMS. It randomly select a row, called a *pivot row*, based on which a query is constructed that must fetch the pivot row. To guarantee that the row is fetched, the testing approach executes a randomly-generated predicate and then modifies it so it evaluates to **TRUE**. While this technique was highly effective, a significant limitation is that its implementation effort is high, since the tool needs to implement all operators and functions that are tested. Furthermore, it can only effectively test `WHERE` clauses, since it validates results based on a single row. Although it can generate `DISTINCT` clauses and `GROUP BY`s, it cannot detect mistakenly fetched duplicate rows, and omitted duplicate rows. PQS can test aggregate functions only when the table contains a single row, which does not meaningfully test their aggregation functionality. The approach proposed in this paper seeks to complement PQS. Query Partitioning can detect bugs in a wider range of features and requires little implementation effort. PQS can provide a ground truth for an important selection of core operators, and thus fill a gap left open by TLP.

NoREC. Non-optimizing Reference Engine Construction (NoREC) was recently proposed by Rigger and Su [2020a] to find *optimization bugs* in DBMS, which are logic bugs that cause the DBMS to incorrectly apply an optimization. The core insight of this approach is that an optimized query can be translated to one that the DBMS cannot effectively optimize. Thus, NoREC is also a metamorphic testing approach. NoREC could also have detected the bug in the motivating example (see Listing 1). Specifically, it would rewrite query ① to another query `SELECT (t0.c0 = t1.c0) IS TRUE FROM t0, t1`. The translated query evaluates the predicate that is taken from the **WHERE** clause of the original query on every row in the table; since only one row is contained, the query would return a single row with a single column whose value is **TRUE**. In practice, the number of **TRUE** values would be summed up using the `SUM()` aggregate function. A predicate must always evaluate to the same value. Thus, it would be expected that the predicate evaluates to **TRUE** in the **WHERE** clause, meaning that the result set of the original query should comprise the row. For this query, this is not the case, and would allow finding the bug. In fact, the original query was optimized by the DBMS to efficiently fetch the data, while the translated query evaluates the predicate on every row, which made the incorrect optimization inapplicable. As with PQS, NoREC has been successful in detecting a wide range of bugs. However, similar to PQS, a significant limitation is that the approach is applicable only to **WHERE** clauses (and partially **GROUP BY** clauses). TLP advances NoREC in two important ways. First, NoREC tackles the test oracle problem by inhibiting DBMS optimizations, while TLP tackles the problem by partitioning a given query, thus, at the conceptual level, they are orthogonal and complement each other. Second, it is unclear how NoREC could be extended to support other features. In particular, aggregate functions are beyond the scope for NoREC, since for these, bugs are due to incorrect computations in the aggregate functions themselves, rather than due to incorrectly-evaluated predicates. We address these limitations through TLP. For example, TLP can detect bugs in aggregate functions by partitioning their computations. Our evaluation results demonstrate these distinct benefits of TLP.

3 APPROACH

Query Partitioning. We envisionage Query Partitioning as a versatile technique. The core idea of our approach is to start from a given query and decompose it to multiple equivalent queries, whose results can be composed to obtain the same result as the original query. We refer to the given query as the *original query*. In the remainder of this paper, we assume that the original query is randomly generated according to the specified format, but it could likewise be given by a user or specified in a test suite. We refer to the multiple queries that are equivalent to the original query as the *partitioning queries*, each of which computes a *partition*. We refer to the operator that combines the partitions as the *composition operator* (denoted by ⋄).

Ternary Logic Query Partitioning. In this paper, we consider only a single instance of the general partitioning strategy idea, namely Ternary Logic Partitioning (TLP). The core idea of the technique is that a predicate on a row or intermediate result must either evaluate to **TRUE**, **FALSE**, or **NULL**. Thus, an original query can be decomposed to three partitioning queries. One partitioning query fetches rows where a predicate p holds, one where it does not hold, and one for which it evaluates to **NULL**. That is, we construct one predicate p, one predicate **NOT** p, and one predicate p IS **NULL**. Each predicate is then used in **WHERE** and **HAVING** clauses. Accordingly, we refer to these predicates as *ternary predicate variants*. Similarly to the original query, we assume this predicate to be randomly generated. In the further, we demonstrate how this idea enables testing **WHERE** clauses, **GROUP BY** clauses, **HAVING** clauses, aggregate functions, and **DISTINCT** queries.

Process. Figure 1 illustrates the process of TLP. Based on an existing database—which is randomly generated in our implementation—a random query Q is generated. We denote the result set of this query as $RS(Q)$ and illustrate a result set using a circle. Based on TLP, we then derive three partitioning queries Q'_p, $Q'_{\neg p}$, and $Q'_{p\ IS\ NULL}$ from Q. Each partitioning query computes a partition of the result, which we denote as $RS(Q'_p)$, $RS(Q'_{\neg p})$, and $RS(Q'_{p\ IS\ NULL})$. Based on the composition operator ⋄, the individual partitions are composed to obtain a result set $RS(Q')$. The equality $RS(Q) = RS(Q')$ must hold. If we find that the result sets differ, a bug in the DBMS is detected.

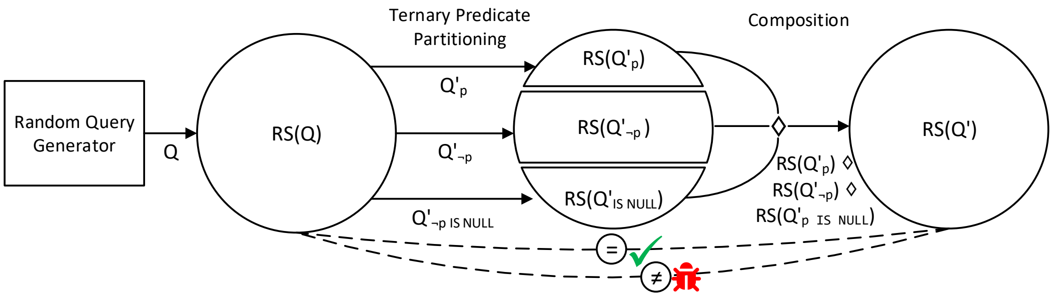

Fig. 1. The idea of Ternary Logic Partitioning (TLP) is to partition a query Q into several partitioning queries Q'_p, $Q'_{\neg p}$, and $Q'_{p\ IS\ NULL}$, whose partitions are composed to form a result set $RS(Q) = RS(Q')$. The dashed lines indicate a comparison between the result sets. The check represents that the result sets match as expected, while the bug represents that the DBMS is affected by a bug, causing the result sets to mismatch.

Intuition on the partitions. Intuitively, the partitioning queries can be considered as computing a subset or a partial bag of Q's result set (*i.e.*, the partitions are subsets or bags of $RS(Q)$). For both the WHERE and HAVING test oracles, ⋄ corresponds to the multiset addition ⊎. For the DISTINCT and GROUP BY oracles, the partitions can contain duplicate values; for these oracles, the ⋄ corresponds to the set union ∪. For the aggregate test oracle, the partitions are not a subset of the original query's result; rather, they correspond to intermediate values. For example, when testing the **MIN**() aggregate function, which computes the minimum, the partitions denote the minimum value of their individual partitions.

Overview. Table 1 shows all the information necessary to fully realize the oracles, which are explained in detail in the subsequent sections. The first column denotes the oracle's name. The second column describes the format of the randomly-generated query Q. The third column describes the format of a partitioning query $Q'_{p_{tern}}$. This query is instantiated with the three ternary predicate variants. The fourth column describes the implementation of the composition operator. Reconsider the motivating example (see Listing 1), which we found using the WHERE oracle, described in the first row of the table. Query Ⓞ corresponds to the format of Q, while each partitioning query in query Ⓟ corresponds to the $Q'_{p_{tern}}$ format. The partitions in query Ⓞ are composed using the **UNION ALL** operator, which corresponds to the $\uplus$ operator.

Query elements. The <columns> placeholder refers to a set of columns, or expressions that are evaluated on each of the rows; this placeholder could also be an asterisk ($_*$), specifying that all columns should be fetched. The <tables> placeholder refers to the tables, from which values are fetched. The <joins> placeholder can refer to any of the joins (such as inner joins, outer joins, left joins, right joins, and natural joins); although we do not propose an exclusive oracle to test joins, we found that the existing oracles also detect bugs in them. The <e> placeholder refers to an arbitrary expression. An element enclosed in square brackets ([]) denotes that the element is optional.

ORDER BYs. A random **ORDER BY** can be generated for each of the partitioning queries. Since our oracles do not validate the ordering of the result, such clauses must not affect the query's result. However, they introduce additional complexity (*e.g.* by causing a DBMS to use an index for sorting [Graefe 2011]), which can help to expose additional bugs. In fact, we found bugs that were only triggered when using an **ORDER BY** clause (see Listing 7). Some DBMS do not allow individual **ORDER BY**s in queries joined using **UNION** or **UNION ALL**; for them, only a single **ORDER BY** might be used when the partitions are composed using **UNION** or **UNION ALL** operators (see below).

Composition operator implementation. Every composition operator either contains a $\uplus$ or $\cup$ operator to compose result sets with, and without removing duplicate rows. The testing tool can implement them by iterating over each partitioning query's result set and collecting the rows using an appropriate data structure—for example, a list for $\uplus$ and a set for $\cup$. Implementing the operator in the testing tool is not applicable when the partition is used for further computations, like in the aggregate oracles. For example, the aggregate MIN oracle computes the minimum value of each partition's minimum value; the minimum value cannot be easily determined by the testing tool, since, for example, the order of strings can depend on **COLLATE** clauses that can be part of the query. For these, a more convenient alternative is to use the **UNION ALL** and **UNION** operators, which implement the operators' semantics in SQL (see Section 2). Using them also tests these operators, and, in fact, we found bugs in their implementation.

3.1 Testing WHERE Clauses

The WHERE oracle tests the correct implementation and optimization of **WHERE** clauses. It is the most basic test oracle. Nevertheless, our evaluation shows that it is the most effective.

Queries. The WHERE oracle assumes an original query that lacks a **WHERE** clause, and constructs partitioning queries with a **WHERE** clause, each of which uses one of the ternary logic predicates. Our intuition is that the original query is unlikely to compute an incorrect result, since it simply fetches all records of a set of tables. In contrast, the partitioning query's **WHERE** clauses might result in the incorrect omission or addition of records.

Intuition on the test oracles. We believe that the WHERE oracle is sufficient to find the majority of bugs that the TLP oracles can detect. While it specifically generates queries to test **WHERE** clauses, it also stresses the implementation of a variety of DBMS components and optimizations [Chaudhuri 1998]. Specifically, we found that this test oracle can find bugs in physical access methods (in

Table 1. Each of the Ternary Logic Partitioning (TLP) oracles is designed to test a specific SQL feature.

Oracle	Q	$Q'_{p_{tern}}$	$\diamond(Q'_p, Q'_{\neg p}, Q'_{p\ NULL})$

WHERE	**SELECT** <columns>
	FROM <tables> [<joins>]
		SELECT <columns>
		FROM <tables> [<joins>]
		WHERE p_{tern}
			$Q'_p \uplus Q'_{\neg p} \uplus Q'_{p\ NULL}$

WHERE Extended	**SELECT** <columns>
	FROM <tables> [<joins>]
	WHERE $p_{existing}$
		SELECT <columns>
		FROM <tables> [<joins>]
		WHERE $p_{existing}$ AND p_{tern}
			$Q'_p \uplus Q'_{\neg p} \uplus Q'_{p\ NULL}$

GROUP BY	**SELECT** <columns>
	FROM <tables> <joins>
	GROUP BY <columns>
		SELECT <columns>
		FROM <tables> <joins>
		WHERE p_{tern}
		GROUP BY <columns>
			$Q'_p \cup Q'_{\neg p} \cup Q'_{p\ NULL}$

HAVING	**SELECT** <columns>
	FROM <tables> <joins>
	[**WHERE** ...]
	[**GROUP BY** ...]
		SELECT <columns>
		FROM <tables> <joins>
		[**WHERE** ...]
		[**GROUP BY** ...]
		HAVING p_{tern}
			$Q'_p \uplus Q'_{\neg p} \uplus Q'_{p\ NULL}$

DISTINCT	**SELECT DISTINCT**
	<columns>
	FROM <tables> <joins>
		SELECT [**DISTINCT**]
		<columns>
		FROM <tables> <joins>
		WHERE p_{tern}
			$Q'_p \cup Q'_{\neg p} \cup Q'_{p\ NULL}$

Aggregate (MIN)	**SELECT MIN**(<e>)
	FROM <tables> [<joins>]
		SELECT MIN(<e>)
		FROM <tables> [<joins>]
		WHERE p_{tern}
			$MIN(Q'_p \uplus Q'_{\neg p} \uplus Q'_{p\ NULL})$

Aggregate (MAX)	**SELECT MAX**(<e>)
	FROM <tables> [<joins>]
		SELECT MAX(<e>)
		FROM <tables> [<joins>]
		WHERE p_{tern}
			$MAX(Q'_p \uplus Q'_{\neg p} \uplus Q'_{p\ NULL})$

Aggregate (SUM)	**SELECT SUM**(<e>)
	FROM <tables> [<joins>]
		SELECT SUM(<e>)
		FROM <tables> [<joins>]
		WHERE p_{tern}
			$SUM(Q'_p \uplus Q'_{\neg p} \uplus Q'_{p\ NULL})$

Aggregate (COUNT)	**SELECT COUNT**(<e>)
	FROM <tables> [<joins>]
		SELECT COUNT(<e>)
		FROM <tables> [<joins>]
		WHERE p_{tern}
			$SUM(Q'_p \uplus Q'_{\neg p} \uplus Q'_{p\ NULL})$

Aggregate (AVG)	**SELECT AVG**(<e>)
	FROM <tables> [<joins>]
		SELECT SUM(<e>) **as** s,
		COUNT(<e>) **as** c
		FROM <tables> [<joins>]
		WHERE p_{tern}
			$\frac{SUM(s(Q'_p \uplus Q'_{\neg p} \uplus Q'_{p\ NULL}))}{SUM(c(Q'_p \uplus Q'_{\neg p} \uplus Q'_{p\ NULL}))}$

particular index scans) [Astrahan et al. 1976], common physical operators [Chaudhuri 1998], join algorithms [Graefe 1993], rewriting of queries [Haas et al. 1989], and general optimizations that are applied to predicates (*e.g.*, algebraic simplifications). We quantify this observation in Section 5.3.

Existing predicates. It might be desirable to create additional test queries based on queries that already have a **WHERE** predicate, for example, when the original query is not randomly generated, but when existing queries from a test suite are used. We propose the **WHERE** Extended oracle for this scenario. Based on an existing **WHERE** clause and a predicate $p_{existing}$, partitioning queries are derived that use the **AND** operator to add an additional ternary variant to the predicate.

Comparison to NoREC. We believe that the WHERE oracle has similar bug-finding capabilities as NoREC (see Section 5.2). Both test oracles focus on testing **WHERE** clauses. NoREC mainly focuses on testing for optimization bugs, by evaluating the predicate on every row, which disables most optimizations. The **WHERE** oracle achieves this by introducing three variants of the query, which are optimized to different degrees. For example, an index might only be applicable for one or two of the partitioning queries, but not all of them, enabling it to also find such optimization bugs.

3.2 Testing Grouping

The `DISTINCT`, `GROUP BY` and `HAVING` test oracles are closely related, as they all test the grouping and filtering of rows. We refer to them collectively as *grouping oracles.*

Queries. The `DISTINCT` oracle is based on the composition operator, which excludes duplicate rows using the ∪ operator. The partitioning queries themselves can thus optionally omit the **DISTINCT** keyword. The `GROUP BY` test oracle, similarly to the `DISTINCT` test oracle, relies on the ∪ operator to exclude duplicate rows. The columns in the **GROUP BY** must correspond to those columns that are fetched. If the **GROUP BY** clause would contain additional columns that are not represented in <columns>, then the additional groups would be invisible for the composition operator. Similarly, if columns that are fetched are not represented in the **GROUP BY** clause, duplicate values would nevertheless be removed by ∪. Note that this might prevent some bugs from being found. The `HAVING` oracle validates that **HAVING** clauses, which are logically applied after the **GROUP BY** is performed, are performed correctly. Thus, unlike the `DISTINCT` and `GROUP BY` oracles, the ternary predicates are used in the **HAVING** clause, rather than in the **WHERE** clause.

Example. Listing 2 gives a representative example for the grouping oracles, specifically for the `DISTINCT` oracle, to illustrate the format of the queries and to give an example of a bug they can detect. The original query Ⓞ contains a **DISTINCT** clause and computes the correct value `{0|0}`. The partitioning queries Ⓟ compute an incorrect result `{0|NULL}`, since the affinity of the view column `c0` is mistakenly discarded—the affinity of a column determines what implicit conversions are performed and is a concept unique to SQLite. Note that when removing the **DISTINCT** clause, the query computes the same incorrect result as the partitioning queries, which is why the `WHERE` test oracle cannot detect this bug. As discussed, the subqueries can optionally discard the **DISTINCT** clause; in this example, either option would have detected the bug.

Listing 2. This simplified `DISTINCT` test case found a bug in SQLite and exemplifies the structure of the queries of the grouping oracles.

```
CREATE TABLE t0(c0 INT);
CREATE VIEW v0(c0) AS SELECT CAST(t0.c0 AS INTEGER) FROM t0;
INSERT INTO t0(c0) VALUES (0);
Ⓞ SELECT DISTINCT * FROM t0 LEFT OUTER JOIN v0 ON v0.c0 >= '0'; -- {0|0}✔
Ⓟ SELECT * FROM t0 LEFT OUTER JOIN v0 ON v0.c0 >= '0' WHERE TRUE UNION
        SELECT * FROM t0 LEFT OUTER JOIN v0 ON v0.c0 >= '0' WHERE NOT TRUE UNION
        SELECT * FROM t0 LEFT OUTER JOIN v0 ON v0.c0 >= '0' WHERE TRUE IS NULL; --
            {0|NULL}🐞
```

3.3 Aggregate Functions

The aggregate query partitioning test oracles are used to test aggregate functions. We consider the most-commonly used aggregate functions **MIN**(), **MAX**(), **COUNT**(), **SUM**(), and **AVG**(). Aggregate functions can be optimized by decomposing the computation and distributing it [Jesus et al. 2015]. We use the core idea of distributing the computation as a basis for testing aggregate functions.

Self-composable aggregate functions. The simplest test oracles for aggregate functions are for self-composable aggregate functions (*i.e.*, `MIN()`, `MAX()`, `SUM()`, and `COUNT()`). Unlike for the oracles introduced above, the partition for aggregate functions is an intermediate result, rather than a subset of the original query's result set. For example, a partition of the `MIN` oracle computes the minimum value of the respective partitioning query. To compute the partitions, an additional step is necessary; for example, for `MIN()`, the overall minimum value must be computed. To this end, another aggregate function can be applied; for example, to compute the overall minimum value, `MIN()` can be applied once more. The aggregate function for the composition operator is not necessarily the same as for the partitioning query. Consider, for example, the `COUNT` oracle. The partitioning queries compute the number of rows in their partition using `COUNT()`. They are then summed up using the `SUM()` aggregate function.

Self-composable aggregate functions example. Listing 3 shows an example for an original query Ⓞ, and the partitioning queries Ⓟ, which is an actual test case generated by the `MAX` oracle, which detected a bug in CockroachDB. In this example, the original query fetched `NULL`, rather than `0`, which was the result set returned by the composed partitioning queries. This bug affected interleaved tables, which are used to implement parent-child relationships between tables, when experimental vectorization features were turned on. The developers explained that an incorrect predicate was used to skip interleaved child rows when performing a reverse scan. Note that the partitioning queries' results must be assigned an alias (`as aggr`), so that the partitions can be composed.

Listing 3. This simplified `MAX()` test case detected a bug in CockroachDB, and exemplifies the structure of the queries for self-composable aggregate functions.

```
SET vectorize=experimental_on;
CREATE TABLE t0(c0 INT);
CREATE TABLE t1(c0 BOOL) INTERLEAVE IN PARENT t0(rowid);
INSERT INTO t0(c0) VALUES (0);
INSERT INTO t1(rowid, c0) VALUES(0, TRUE);
Ⓞ SELECT MAX(t1.rowid) FROM t1; -- {NULL}🐞
Ⓟ SELECT MAX(aggr) FROM (
      SELECT MAX(t1.rowid) as aggr FROM t1 WHERE '+' >= t1.c0 UNION ALL
      SELECT MAX(t1.rowid) as aggr FROM t1 WHERE '+' >= t1.c0 UNION ALL
      SELECT MAX(t1.rowid) as aggr FROM t1 WHERE '+' >= t1.c0
); -- {0}✓
```

Other composable aggregate functions. For aggregate functions that are not self-composable, but composable, such as `AVG()`, we can compute the results using a result tuple, rather than a single value. For example, to compute `AVG()`, we utilize that $AVG(Q)$ corresponds to $SUM(Q)/COUNT(Q)$. Accordingly, each partitioning query computes a tuple $(SUM(Q_p), COUNT(Q_p))$, which is then composed by dividing the sum of the first tuple values by the sum of the second.

Composable aggregate function example. Listing 4 gives a concrete example on a `AVG` oracle test case that found a bug in DuckDB. Query Ⓞ shows an original query that computes the `AVG()` of the values contained in column `c0`. Each partitioning query Ⓟ computes two values, one being the sum (aliased as `s`) and one being the count of values in `c0` (aliased as `c`). The expression `SUM(s)/SUM(c)` is associated with the composition operator; it divides the accumulated sums with the accumulated counts. For this test case, DuckDB computed the correct result for the original query. For the partitioning queries, only the first aggregate query fetches a row, which is expected. However, the addition of 9223372036854775807 and 2 in $SUM()$ overflowed, which was an unexpected result and caused a silent wraparound. The bug was confirmed as a real bug. However, the developers were already aware of it, and had not addressed it, since addressing this bug without significant performance impact was non-trivial. After we reported it, they nevertheless decided to fix it.

Listing 4. This simplified AVG test case demonstrates a bug in DuckDB, and shows the structure of the queries for composable, but not self-composable aggregate functions.

```
CREATE TABLE t0(c0 BIGINT);
INSERT INTO t0(c0) VALUES (2);
INSERT INTO t0(c0) VALUES (9223372036854775807);
Ⓞ SELECT AVG(t0.c0) FROM t0; -- {4.611686018427388e+18}✔
Ⓟ SELECT SUM(s)/SUM(c) FROM (
        SELECT SUM(t0.c0) AS s, COUNT(t0.c0) AS c FROM t0 WHERE c0 UNION ALL
        SELECT SUM(t0.c0) AS s, COUNT(t0.c0) AS c FROM t0 WHERE NOT c0 UNION ALL
        SELECT SUM(t0.c0) AS s, COUNT(t0.c0) AS c FROM t0 WHERE c0 IS NULL
); -- {-4611686018427387903}🐞
```

Commutativity. All the aggregate functions that we considered are commutative. For our purpose, we also assume `SUM()` and `AVG()` to be commutative, although the processing order matters for floating-point numbers. To account for rounding errors caused by this, we compare floating-point numbers in the result sets using an epsilon. Other non-commutative aggregate functions, such as `GROUP_CONCAT()`, which concatenates strings, exist. In order to support these, an operator-specific comparator could be implemented. For a example, a comparator for `GROUP_CONCAT()` could split the concatenated string by its delimiter(s), sort the tokens, and use the sorted tokens for comparison. Such an implementation would be more tedious compared to the other test oracles. Furthermore, non-commutative functions provide less optimization potential for the DBMS. Thus, we did not consider non-commutative functions further in our work.

4 SELECTED BUGS

This section gives an overview of *interesting bugs* that we found using TLP. This selection is necessarily biased, and we sought to demonstrate the range of different bugs that the individual oracles detected. For brevity, we show only reduced test cases that demonstrate the underlying core problem, rather than the original and partitioning queries that found the bugs.

4.1 WHERE clauses

This section presents bugs detected by the WHERE oracle. Unless noted otherwise, these bugs can also be detected by NoREC and PQS. Note that in Section 5.2, we systematically investigate the relationship between the WHERE oracle and NoREC.

MySQL comparison bug. Listing 5 shows a bug where a comparison of numbers yielded an incorrect result. The comparison `0.9 > t0.c0` should evaluate to `TRUE` for `c0=0` and fetch the row in `t0`. However, MySQL failed to fetch the row. This is one of multiple basic bugs that we found in MySQL. We still consider it interesting, since it shows that also mature DBMS are prone to such bugs.

Listing 5. MySQL incorrectly evaluated the comparison and failed to fetch the row.

```
CREATE TABLE t0(c0 INT);
INSERT INTO t0(c0) VALUES (0);
SELECT * FROM t0 WHERE 0.9 > t0.c0; -- {0}✔ {}🐞
```

TiDB comparison bug. We found a bug in TiDB where fetching from a view unexpectedly omitted a row (see Listing 6). The `WHERE` clause should evaluate to `TRUE` and fetch a row, since it refers to the constant value 1 in the view. However, TiDB unexpectedly did not fetch a row. The bug was classified as a P1 bug, which is the second highest severity category. We believe that this bug is interesting, since it demonstrates that our approach can detect bugs in views, without specifically aiming to test them.

Listing 6. TiDB failed to fetch a row from a view.

```
CREATE TABLE t0(c0 INT);
CREATE VIEW v0(c0, c1) AS SELECT t0.c0, 1 FROM t0;
INSERT INTO t0 VALUES (0);
SELECT v0.c0 FROM v0, t0 WHERE v0.c1; -- {0}✓ {}🐞
```

ORDER BY affects a query's result. We found a bug in CockroachDB, where a value was unexpectedly represented using the E notation (see Listing 7). Specifically, while the default row engine encodes the fetched decimal value as `1819487610`, the vector-based engine, which was used for the partitioning queries, represented the value as `1.81948761E+9`. While this was confirmed as a bug, it was not deemed to be very important, considering that both represent the same value. However, we believe that this bug is interesting, since it demonstrates that an **ORDER BY** can incorrectly influence a query's result.

Listing 7. The ORDER BY clause affected the representation of the decimal value 1819487610 when using the vector-based execution engine in CockroachDB.

```
SET SESSION VECTORIZE=on;
CREATE TABLE t0 (c0 DECIMAL PRIMARY KEY, c1 INT UNIQUE);
INSERT INTO t0(c0) VALUES (1819487610);
SELECT t0.c0 FROM t0 ORDER by t0.c1; -- {1819487610}✓ {1.81948761E+9}🐞
```

Missing error for invalid regular expression. We found a bug in CockroachDB where an invalid regular expression caused a **SELECT** to retrieve an empty result set, rather than printing an error message (see Listing 8). We found this bug because also the other partitioning queries did not fetch any rows. Both PQS and NoREC could not detect such bugs, since for these approaches, the original query would result in the expected error above. Rigger and Su [2020a] specifically explain that NoREC cannot detect errors due to nondeterminism in the evaluation of queries.

Listing 8. Rather than exiting with an error, CockroachDB returned an empty result set for this query.

```
CREATE TABLE t0(c0 INT);
CREATE VIEW v0(c0) AS SELECT COUNT_ROWS() FROM t0;
SELECT * FROM v0 WHERE '' !~ '+'; -- error parsing regexp: missing argument to
    repetition operator: +✓ {}🐞
```

4.2 Grouping Bugs

This section presents bugs that were detected by the GROUP BY, HAVING, and DISTINCT oracles.

GROUP BY disregards COLLATE. We found a bug in DuckDB, where the **GROUP BY** operator disregarded a **COLLATE** NOCASE (see Listing 9). Note that a **COLLATE** clause controls the behavior of comparisons for strings; in this example, it specifies that string comparisons should be performed without considering the case of the strings. While the **SELECT** was expected to return a result set containing either `'a'` or `'A'`, it fetched both. The **GROUP BY** oracle detected this bug, since, unlike the **GROUP BY** operator, the **UNION** operator respected the **COLLATE**. This bug is interesting, since it demonstrates a basic bug in the operator itself, rather than an optimization bug, to which NoREC is limited. However, we found this bug shortly after **COLLATE**s were merged to `master`, and before this feature was released, suggesting that this feature was not yet thoroughly tested.

Listing 9. The GROUP BY operator disregarded that c0 has a COLLATE NOCASE in DuckDB.

```
CREATE TABLE t0(c0 VARCHAR COLLATE NOCASE);
INSERT INTO t0(c0) VALUES ('a'), ('A');
SELECT t0.c0 FROM t0 GROUP BY t0.c0; -- {'a'} or {'A'}✓ {'a', 'A'}🐞
```

Incorrect VARIANCE(0) optimization. We found a bug in CockroachDB where `VARIANCE(0) IS` **NULL** was unexpectedly optimized to **FALSE** (see Listing 10). Interestingly, `VARIANCE(0)` evaluates to **NULL** if the table contains zero or one rows; if the table contains at least two rows, it evaluates to `0`. The optimization was thus incorrect for this case, where the table contained only one row. We believe that this case is interesting, since aggregate functions cannot be used in **WHERE** clauses, so although this is an optimization bug, it could not have been found by NoREC.

Listing 10. CockroachDB unexpectedly optimized VARIANCE(0) to FALSE.

```
CREATE TABLE t0(c0 INT);
INSERT INTO t0(c0) VALUES(0);
SELECT t0.c0 FROM t0 GROUP BY t0.c0 HAVING NOT (VARIANCE(0) IS NULL); --{}✓ {0}🐞
```

Non-deterministic MAX(). We found a bug in DuckDB, where a complex query using **GROUP BY** and **HAVING** clauses, as well as **UNION** resulted in a nondeterministic result (see Listing 11). As explained by the developers, this bug was caused since non-inlined strings were not being properly copied into the hash table when stored as **MAX**() values. Since this lead to a user-after-free error, this bug might have also been detected by undefined-behavior checkers [Regehr 2010; Stepanov and Serebryany 2015]. We believe that this bug is interesting nevertheless, since it demonstrates the range of bugs that TLP can detect.

Listing 11. DuckDB nondeterministically fetched two and three rows for this query.

```
CREATE TABLE t0(c0 INT);
CREATE TABLE t1(c0 VARCHAR);
INSERT INTO t1 VALUES (0.9201898334673894), (0);
INSERT INTO t0 VALUES (0);
SELECT * FROM t0, t1 GROUP BY t0.c0, t1.c0 HAVING t1.c0!=MAX(t1.c0) UNION ALL
    SELECT * FROM t0, t1 GROUP BY t0.c0, t1.c0 HAVING NOT t1.c0>MAX(t1.c0); --
    nondeterministic result🐞
```

Non-deterministic GROUP BY. We found a bug in TiDB, where a **SELECT** nondeterministically fetched a duplicate row (see Listing 12). We could only reproduce the bug with a large number of rows; note that we removed the **INSERT**s from the listing for brevity. We believe that is likely a bug that is caused by a race condition. TiDB is written in Go, for which race detectors seem to exist, indicating that such a bug might have been found by them. However, race condition checkers are known to be slow [Serebryany et al. 2011], and TLP might be viable and cheaper alternative to identify test cases that trigger race conditions.

Listing 12. TiDB computed a non-deterministic result for this query.

```
CREATE TABLE t0(c0 INT, c1 INT);
CREATE TABLE t1(c0 INT, c1 INT);
CREATE TABLE t2(c0 INT, c1 INT);
-- 27 INSERTS
ANALYZE TABLE t1, t2;
SELECT t1.c0 LIKE t1.c0 FROM t1, t2, t0 GROUP BY t1.c0 LIKE t1.c0; --
    nondeterministic result🐞
```

4.3 Aggregate Bugs

Similar to grouping bugs, aggregate functions are interesting since they cannot be detected by neither NoREC nor PQS.

MAX() and UTF-16 bug. We found a bug in SQLite, where `MAX()` computed an incorrect result for the ordering of UTF-16 strings and non-ASCII characters (see Listing 13). The SQLite developers explained that SQLite was incorrectly using the UTF-8 collating sequence for some, but not all expressions in database having a UTF16LE encoding. Although this bug might seem obscure, we believe that it is interesting, because it would go likely undetected by users, but result in unexpected results when an application relies on it.

Listing 13. SQLite computed an unexpected ordering for special non-ASCII characters and UTF-16LE encoding.

```
PRAGMA encoding = 'UTF-16';
CREATE TABLE t0(c0 TEXT);
INSERT INTO t0(c0) VALUES ('□'), (1);
SELECT MAX(CASE 1 WHEN 1 THEN t0.c0 END) FROM t0; -- {0}✔ {'□'}🐞
```

SUM() optimization. Listing 14 demonstrates an optimization bug in DuckDB. As explained by the developers, to sum up the constants, an optimization `sum += input * count` was applied, where `input` refers to the constant `-1`. Since `count` was declared as an unsigned integer, the result was cast to an unsigned number, resulting in an underflow [Dietz et al. 2012]. This finding demonstrates that also aggregate functions are affected by optimization bugs, which NoREC is unable to find.

Listing 14. DuckDB computed an incorrect result due to an optimization that summed up constants by using an unsigned, rather than a signed integer.

```
CREATE TABLE t0 (c0 INT);
INSERT INTO t0 VALUES (0);
SELECT SUM(-1) FROM t0; -- {-1}✔ {1.8446744073709552e+19}🐞
```

MIN() initialization bug. Listing 15 demonstrates an bug in `MIN()` in DuckDB. The culprit was that the minimum value of the domain, -2^{63} for integers, was used to indicate whether a minimum value has been set. Since the expression `CAST(c0 as BIGINT)<< 32` sets the minimum value for `c0=-1`, the implementation mistakenly assumed that no minimum value was set, and returned `NULL`.

Listing 15. DuckDB assumed that no `MIN()` value was set, since the minimum value corresponds to -2^{63}.

```
CREATE TABLE t0(c0 INT);
INSERT INTO t0 VALUES (-1);
SELECT MIN(CAST(c0 AS BIGINT) << 63) FROM t0; -- {-9223372036854775808}✔ {NULL}🐞
```

5 EVALUATION

We evaluated both the effectiveness and generality of TLP in finding bugs, compare it to NoREC, and investigate the overlap between the individual test oracles.

Table 2. We tested a diverse set of popular and emerging DBMS; all numbers are the latest as of May 2020.

DBMS	Popularity Rank			LOC [2]	First Release	Kind	Tested By
	DB-Engines	Stack Over-flow	GitHub Stars [1]				
SQLite	9	4	1.5k	0.3M	2000	Embedded, OLTP	PQS, NoREC
MySQL	2	1	5.0k	3.8M	1995	Traditional	PQS
PostgreSQL	4	2	6.3k	1.4M	1996	Traditional	PQS, NoREC
CockroachDB	77	-	17.7k	1.1M	2015	NewSQL	NoREC
TiDB	118	-	23.1K	0.8M	2017	NewSQL	PQS[3]
DuckDB	-	-	0.5k	59k	2018	Embedded, OLAP	-

Implementation. We implemented our approach in SQLancer, in which also PQS and NoREC were implemented. Since SQLancer did not support the generation of databases and queries for TiDB and DuckDB, we added support for these systems. Furthermore, we implemented the generation of many small, previously unsupported features in SQLancer (*e.g.*, generating arrays and array operations in CockroachDB). The test oracles are implemented in about 500 LOC for each DBMS under test. Note that our implementation is available at *https://github.com/sqlancer.*

Tested DBMS. In our evaluation, we considered six popular and widely-used DBMS with a wide range of characteristics to demonstrate the generality of TLP (see Table 2). SQLite [2020] and DuckDB [Raasveldt and Mühleisen 2020] are both embedded DBMS, meaning that they run within an application's process. Traditional systems like MySQL [2020] and PostgreSQL [2020] are standalone, meaning that they run in a dedicated process. NewSQL systems like CockroachDB [Cockroach Labs 2020] and TiDB [PingCAP 2020] are distributed relational DBMS, which aim to provide a high degree of scalability by splitting up the database [Pavlo and Aslett 2016]; however, we tested only their SQL component. Online Transactional Processing (OLTP) workloads are those that consist of frequent inserts, updates, and deletes. In contrast, Online Analytical Processing (OLAP) workloads typically involve complex queries with aggregates. Traditional systems, NewSQL systems, and SQLite are mostly optimized towards OLTP workloads. DuckDB is a representative of an OLAP systems, and stores its data column-wise. CockroachDB and TIDB are mainly developed commercially (by Cockroach Labs and PingCAP); they provide an open version of their DBMS, which we tested, on GitHub. DuckDB has been developed by a research group, but "*is intended to be a stable and mature database system.*" SQLite is developed by a small development team lead by D. Richard Hipp. MySQL has open-source contributors, and is also developed by Oracle. PostgreSQL is backed by open-source contributors.

5.1 Effectiveness

Study methodology and challenges. We started testing the DBMS while implementing our approach, and tested them in a period of roughly three months. A significant factor limiting our bug-finding efforts were duplicate test cases for bugs. For a single bug, SQLancer typically generated many test cases that would trigger it, making it infeasible to filter out such test cases manually, which was also observed by Rigger and Su [2020a,b]. While automatic bug prioritization approaches were

[1] For PostgreSQL, MySQL, and SQLite, only (inofficial) GitHub mirrors are available.

[2] These numbers are not accurate, but represent a best effort estimate. We omitted counting tests, where this was possible (using `cloc`). For TiDB, we counted the repositories of PD, TiKV, and TiDB, which are all necessary to run TiDB.

[3] PingCAP implemented PQS for TiDB; for the other DBMS, the approaches were implemented as part of the evaluation of the respective papers [Rigger and Su 2020a,b].

proposed to the need to compilers [Chen et al. 2013], applying them for DBMS would be more challenging and slow due to needing to install, set up, and stop many versions of a single DBMS. To address this, we typically avoided the generation of features that induce already known bugs; however, this was not always possible—for example, when we could not discover the necessary conditions to reproduce the bug—or restricted the bug-finding capabilities significantly (*e.g.*, by avoiding the generation of comparisons). When we found a bug, we first automatically reduced it [Regehr et al. 2012], to then manually produce a minimal test case that demonstrated the underlying bug. Before reporting a bug, we checked the public bug trackers for similar bugs, to avoid creating duplicate bug reports.

Oracle implementation. Since the WHERE oracle is the simplest oracle, we implemented it first for every DBMS. We found that the other test oracles detected bugs that also the WHERE oracle could find. Since the other oracle's test cases were typically more complex than the ones generated by the WHERE oracle (*e.g.* the GROUP BY oracle could detect many of the same bugs, but by generating redundant **GROUP BY**s), it was not desirable to use them before the WHERE oracle's bug-finding saturated. Consequently, we implemented the other oracles only for DBMS for which the WHERE oracle saturated, namely SQLite, PostgreSQL, DuckDB, and CockroachDB. For both TiDB and MySQL, we omitted reporting a number of suspected bugs found by the WHERE oracle, due to the large number of open bugs. For TiDB, 35 bugs were verified, but have not yet been addressed. For MySQL, 6 bugs were confirmed, but not fixed. Many of the MySQL bugs were rather basic (*e.g.* see Listing 5), which prevented us from testing MySQL more comprehensively. Furthermore, we found a large number of open bugs in its bug tracker; in fact, even a large number of the bugs reported by the authors of PQS have not yet been addressed. In addition, MySQL has a closed development process, with only the release versions being publicly available. These appear every 2–3 months. We found one bug in MySQL version 8.0.19, which was fixed quickly, but will appear only in MySQL 8.0.21 (*i.e.*, potentially half a year later). We implemented the WHERE Extended oracle for only one DBMS—CockroachDB—after the simple WHERE oracle could not find additional bugs. It did not detect any additional bugs, which is expected; as explained in Section 3.1, this oracle is mainly useful to utilize an existing test suite that contains queries that have **WHERE** clauses.

Found Bugs. Table 3 shows the number of bugs we found and the status of the corresponding bug reports. We opened 181 bug reports, 175 which were either fixed or confirmed by the developers. 123 bugs have been fixed, which demonstrates that the DBMS developers considered the majority of the bugs to be important. Almost all bugs were addressed by code changes; only 1 bug was addressed by a documentation change. The behavior in 3 bug reports was surprisingly considered to be intended by the bug verifiers of MySQL; we discuss one of them in detail below. We opened only 3 duplicate bug reports, as we carefully checked the bug tracker for similar bugs. We could comprehensively test SQLite, CockroachDB, PostgreSQL, and DuckDB. That is, we were not restricted by any open bug reports that preventing us from testing by making it difficult to filter out duplicate test cases for bugs. For TiDB and MySQL, we stopped testing due to the large number of open bug reports. We found more bugs in DuckDB and TiDB than in the other DBMS, since the other DBMS were comprehensively tested by NoREC and PQS (see Table 2). We did not find any bugs in PostgreSQL, which is why omitted this DBMS from the table. This is not surprising. PQS detected only one logic bug in it, and NoREC did not detect any logic bugs.

Test Oracles. Table 4 shows how many bugs each individual test oracle detected. In total, we found 77 logic bugs. The WHERE oracle detected 60 bugs, suggesting that—even though it is the conceptually simplest oracle—it is the most effective one. For DBMS that were intensively tested, like

Table 3. We found 175 previously unknown bugs, 123 of which have been fixed.

			Closed	
DBMS	Fixed	Verified	Intended	Duplicate
SQLite	4	0	0	0
MySQL	1	6	3	0
CockroachDB	22	9	0	0
TiDB	26	35	0	1
DuckDB	71	1	0	2

Table 4. We found 60 bugs with the `WHERE` oracle, 10 with the aggregate oracle, 3 with the `HAVING` oracle, and the others by internal DBMS errors and crashes.

	Query Partitioning Oracle						
DBMS	`WHERE`	Aggregate	`GROUP BY`	`HAVING`	`DISTINCT`	Error	Crash
SQLite	0	3	0	0	1	0	0
CockroachDB	3	3	0	1	0	22	2
TiDB	29	0	1	0	0	27	4
MySQL	7	0	0	0	0	0	0
DuckDB	21	4	1	2	1	13	19

SQLite and CockroachDB, this oracle was less effective. In Section 5.2, we will thus closely investigate the relationship between NoREC and the query partitioning `WHERE` oracle. The other oracles detected 17 bugs in total; although many of these bugs were serious, the number of found bugs is low when compared to the bugs found by the `WHERE` oracle. Our analysis suggests that the features tested by these oracles relies mostly on functionality in the DBMS that is tested by `WHERE` oracle; in Section 5.3, we investigate this hypothesis based on coverage information. Besides logic bugs, we found 25 crash bugs and 62 error bugs. Crash bugs refer to process crashes (*e.g.*, a memory error resulting in a `SEGFAULT`). Error bugs were due to unexpected errors in the DBMS (*e.g.*, internal errors printing a stack trace). The higher number of crash bugs in DuckDB is explained by us using the debug build for testing, which resulted in assertion violations, which accounted for 11 of the crash bugs. While the developer also appreciated those bug reports, finding them was not a goal for us, since they could have been found by existing approaches, such as fuzzers. We report these numbers nevertheless, to put the numbers of found logic bugs into relation. As with PQS and NoREC, we found larger number of error and crash bugs than logic bugs.

Soundness. In theory, our approach should not be affected by false positives (*i.e.*, when SQLancer reports a bug, it is always a real bug). However, the MySQL bug verifiers considered 3 of our bug reports as false positives. For example, consider the test case in Listing 16. The `WHERE` oracle found an inconsistency in the result, since neither of the partitioning queries fetched a row, while the original query fetched the row with `c0=0`. While reducing the bug, we found strong evidence that led us to believe that this was indeed a bug. First, rewriting the query to evaluate the predicate indicated that the predicate should evaluate **`TRUE`**, so also NoREC would have considered it to be a bug. Second, the query's behavior changes when omitting the **`UNIQUE`** constraint and yields the result we would expect; this is unexpected, because an index should never affect a query's result. Third, an earlier version of MySQL computed the result we expected. Fourth, TiDB, which strives to be compatible with MySQL, computed the result we would expect. Despite these, the report was

Listing 16. The report associated with this test was considered a false positive by the MySQL bug verifiers.

```
CREATE TABLE t0(c0 DECIMAL UNIQUE);
INSERT INTO t0(c0) VALUES(0);
SELECT * FROM t0 WHERE '' BETWEEN t0.c0 AND t0.c0; -- {0}✔ {}🐞
```

closed, based on the argument that *Oracle !8c* computes the same result. After we further inquired, a second bug verifier subsequently elaborated that an empty `STRING` cannot represent a valid **`DECIMAL`** value, referring to the SQL standard. While indeed a warning that the empty string is not a valid **`DECIMAL`** is printed, such warnings are printed for many other queries too, which do not violate the TLP assumptions. Thus, we still believe that TLP is sound, and does not result in false positives.

5.2 Comparison with NoREC

We studied how NoREC relates to TLP. Both NoREC and TLP are metamorphic oracles and have similar advantages (*e.g.*, the small implementation effort required to realize them) as well as the same disadvantage (*i.e.*, they cannot establish a ground truth). We did not compare to PQS, which is complementary to TLP and NoREC, and whose advantages and disadvantages were already studied [Rigger and Su 2020a]. Both PQS and NoREC are mostly limited to finding bugs in **`WHERE`** clauses and are not applicable to the other features TLP can test (see Section 2). Consequently, we compare only the TLP `WHERE` oracle with NoREC.

Methodology. Fairly comparing the two techniques is challenging. Optimally, we could apply each technique to the same DBMS and compare the number of distinct bugs that they find. However, determining whether a test case triggers a specific bug would be difficult and labor-intensive to determine [Marcozzi et al. 2019]. Given that NoREC had been used to test two DBMS before we tested them using TLP—SQLite and CockroachDB—analyzing any additional bugs that the `WHERE` oracle found gives an insight into what additional bugs it can find. Similarly, for DBMS in which the `WHERE` oracle did not find any additional bugs, NoREC could be applied to validate whether it can find any additional bugs. Given that DuckDB is the only DBMS that had not been tested by NoREC, and on which our testing efforts saturated, we implemented and tested NoREC only on this DBMS. In addition, we sought to give an estimate on the oracles' overlap based on a manual analysis of the found bugs. Specifically, we tried to translate a NoREC test case to a `WHERE` oracle test case and vica versa, by following a similar methodology as for the comparison of NoREC and PQS [Rigger and Su 2020a]. For the majority of cases, this is straightforward. To translate a NoREC test case to a `WHERE` oracle test case, we can take the original query with a `WHERE` clause, and create the two other partitioning queries by assuming the `WHERE` clause predicate to be the randomly-generated predicate based on which the ternary variants are derived. To obtain the original query, the `WHERE` clause must be removed. Similarly, to translate a `WHERE` oracle test case to a NoREC test case, one of the partitioning queries can be assumed as the original query for NoREC. In fact, this was not necessary for many queries, as we typically used a NoREC test case to demonstrate the underlying bug, which is more compact than a TLP test case. The limitation of this manual analysis is that for bugs for which we cannot derive an equivalent test case, we cannot necessarily conclude that no such test case exists, because a different test case might trigger the same underlying bug.

Additional `WHERE` *bugs in DBMS tested by NoREC.* SQLite and CockroachDB were extensively tested by NoREC, and we found 3 additional bugs in them using the `WHERE` oracle (all in CockroachDB). In a first step, we closely analyzed these bugs to determine whether NoREC could have found them, using the methodology to translate test cases described above. One bug could have been found directly by NoREC; we speculate that it was not found because the test case triggering the bug relied on the **INTERVAL** data type, which we added to SQLancer, and which was not present when NoREC was evaluated. The bug in Listing 7 could have been found by NoREC, but only if the content of the records is fetched in the translated query, which was described as unnecessary by Rigger and Su [2020a]. The bug in Listing 8 could not have been found by NoREC, since the translated query results in the expected error, rather than yielding an unexpected result.

Additional NoREC bugs in DBMS tested by TLP WHERE*.* DuckDB is the only DBMS for which our bug-finding efforts saturated, and which has not yet been tested by NoREC. Thus, we implemented NoREC for this DBMS to determine whether NoREC could find any bugs in this system. Note that DuckDB does not provide the `IS` **TRUE** and `IS` **FALSE** operators, which are used in the translated query that the DBMS is unlikely to optimize. However, this is not problematic, since the translation can be implemented using other operators. Specifically, an original, potentially optimized query with a predicate `p` can be translated to a query `SELECT SUM(count) FROM (SELECT (p IS NOT NULL AND p)::INT as count FROM <tables>)`, so that the size of the original query's result set must be equal to the count obtained by the second query. The more complex translated query does not hinder the effectiveness of NoREC's bug-finding capabilities, since, as with the original approach, the expression has to be evaluated on every record of the target tables, disabling many optimizations. Overall, we did not find any bugs using NoREC on DuckDB. Note that we verified that NoREC could have detected bugs that were found by the WHERE oracle.

Manual analysis of the NoREC bugs. In total, NoREC found 50 bugs. We could mechanically translate 42 NoREC test cases so that the bugs could have been found using the WHERE oracle. For 8 test cases, a mechanic translation as described above was not possible. We identified two root causes for this. The first one was that NoREC detected the bug in an aggregate function that was used to efficiently sum up for how many records a predicate evaluates to TRUE in the translated, unoptimized query, and affected 4 cases. We speculate that these bugs might have been found by one of the TLP aggregate oracles. The second root cause was that the bug was unexpectedly triggered in the translated, unoptimized query, which evaluates the predicate on every row, which affected 4 cases. We believe that the WHERE oracle might overlook these bugs, since it does not compare to which value a predicate is evaluated when used in a different context.

Manual analysis of the TLP WHERE *bugs.* We analyzed all 60 bugs found by the WHERE oracle (counting also the 3 bugs described above). For 48 bugs, we could mechanically derive NoREC test cases. In 5 of these cases, comparing the record count was insufficient to detect the bug; also the contents had to be compared, contrary to prior suggestions [Rigger and Su 2020a]. For the other 12 bugs, it is doubtful that NoREC could have detected them. 3 test cases triggered bugs related to joins and did not require a WHERE clause. Although the WHERE clauses were redundantly generated by the WHERE oracle, it detected these bugs, because the overall number of fetched rows mismatched. 3 test cases triggered bugs in operators, both in NoREC's unoptimized and optimized case. Furthermore, we found 1 bug that was triggered in the UNION operator, which is out-of-scope for NoREC. 1 bug was due to a hint to the query optimizer, which also took effect when used in the translated, unoptimized query, but not in all of the partitioning queries. As mentioned above, one test case resulted in an incorrect result, rather than an error. 3 test cases induced undefined behavior, but did not result in an unexpected result when using NoREC.

5.3 Test Oracle Coverage

During our experiments, we found that different oracles can detect the same underlying bugs in a number of cases, which is an expected behavior. For example, the WHERE oracle specifically aims for testing **WHERE** clauses, but also the subsequent oracles generate **WHERE** clauses, and thus might

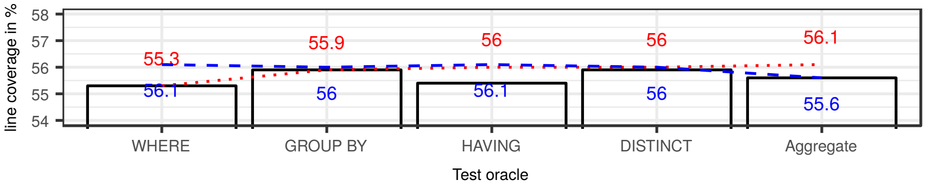

Fig. 2. The bar plots present the coverages of the individual oracles. The red, dotted line and the numbers above it denote the accumulated coverage of the oracles to the left side. The blue, dashed line and the numbers below it denote the same for the oracles to the right side.

detect bugs in their handling. However, subsequent oracles are not guaranteed to find all bugs; for example, the `GROUP BY` oracle might overlook bugs in the handling of **`WHERE`** since an optimization might no longer be applicable when using a **`GROUP BY`**. Furthermore, it would be preferable to use the `WHERE` oracle even for bugs that also the `GROUP BY` oracle can find, since developers typically strive to understand a bug based on a minimal example, where redundant **`GROUP BY`** clauses would slow down triaging and the reduction of the bug, presenting an impediment.

To investigate the overlap quantitatively, we measured the coverage of individual and combined oracles on DuckDB. DuckDB is a good choice for this, since we tested this DBMS comprehensively, and since every oracle found bugs that were not found by the other oracles. Figure 2 displays the line coverage, when running each of the 15 configurations for 10 hours. The barplots show the coverage of the individual oracles. The dotted red line, which rises starting from the left, illustrates the aggregated coverage by summing up all the coverage of the oracles to the left. The dashed blue line, which rises starting from the right, illustrates the same for all oracles to the right.

The maximum coverage that is achieved by utilizing all test oracles is 56.1%. The coverage is rather low, because we did not test components such as subqueries, window functions, transactions, and sequences as well as due to code that is never executed (*e.g.*, due to external dependencies). In comparison, PQS achieved only a coverage reaching from 23.7% to 43.0%. By generating databases alone, already a test coverage of 48.3% is achieved. Each test oracle achieves a similar coverage; the range of test coverages is 0.6% (*i.e.*, reaching from 55.3% to 55.9%). When using test oracles in combination, a small coverage increase can be observed, independent from in which order oracles are combined. However, the `HAVING` oracle seems to decrease the coverage, presumably since it lowers the throughput of the other oracles. Overall, we believe that these findings confirm our intuition that the oracles stress a large common part of the DBMS. Nevertheless, a coverage increase can be observed when adding additional oracles, and, indeed each test oracle found unique bugs.

Despite this evidence that there is a large overlap, it should be noted that coverage information provides only limited insights for DBMS. Jung et al. [2019] found that the core components of DBMS have achieve a coverage of >95% already after running tens of queries. Rigger and Su [2020a] argued for NoREC that coverage information is not insightful, and that they found many bugs in SQLite despite its impressive test suite, which provides 100% MC/DC coverage.

6 DISCUSSION

Bug importance. It is difficult to measure the importance of the bugs we found. The developers of the DBMS we tested explicitly told us that they appreciated our bug-finding efforts, and considered many of the bugs to be important. For example, an engineering manager from Cockroach Labs wrote on a social media platform that we are "*doing the database industry a great service. Thank you!*". Similarly, the most-contributing committer to DuckDB told us: *This work is tremendously helpful for us, and I imagine anyone working on a DBMS. Usually these bugs would be slowly found by* *users over the years, not only negatively affecting the experience of those users but also requiring much more effort to debug and reproduce [...]. For us especially it is extremely helpful because we have not yet gone through decades of users using the system, so this testing allows us to take a massive shortcut and squeeze out many bugs that would otherwise be found by users.* PingCAP started a bug bounty program for a release candidate of their DBMS TiDB, while we were testing it. As part of this, PingCAP also assigned severities to our bug reports, reaching from *P0* (for the most serious issues) to *P3* (documentation bugs). We reported 28 bugs as part of this program. While, based on the bug-bounty guidelines, incorrect query results should result in a *P0* classification, PingCAP updated the guidelines after we reported the first batch of bugs, to reserve them the right to downgrade bugs, to which we agreed. Consequently, 22 bugs were classified as *P1*, and 6 as *P2*, that is, the second-highest and third-highest severities, demonstrating that the bugs we found were deemed important. In fact, we could redeem the points we received to obtain more than 100 T-shirts.

Found bugs. As shown in Table 4, most bugs that we found were crash and error bugs. We do not consider these a contribution of this paper, since they could have also been found by fuzzers and other testing approaches. We list them merely for completeness, and since the give insight on the distribution of errors. That we found more errors and crash bugs might indicate that these are more common, or easier to find than logic bugs. Although the DBMS developers also greatly valued these bugs, we consider logic bugs to be more dangerous. For error and crash bugs, users of the DBMS obtain direct feedback that the query failed (*e.g.*, since the process exits with an error). For logic bugs, however, errors might go unnoticed.

NoREC and PQS. Compared to NoREC and PQS, TLP can detect bugs in **`GROUP BY`** clauses, **`DISTINCT`** queries, **`HAVING`** clauses, and aggregate functions. PQS and NoREC are not applicable for testing most of these features, except partially in corner cases (*e.g.*, when a table contains only a single row, aggregate functions can partially be tested by PQS). TLP is a metamorphic testing approach, and similarly to NoREC, it cannot establish a ground truth (*i.e.*, an operator or function might be consistently behave incorrectly, so that no bugs can be detected). In fact, due to this, Rigger and Su [2020a] found that NoREC can detect only about half of the bugs that PQS can find. Thus, TLP is complementary to PQS, and not a replacement for it. The `WHERE` oracle overlaps with NoREC as demonstrated in Section 5.2. Our manual analysis suggest that the `WHERE` oracle can find 12 bugs that NoREC can find, and that NoREC can find 8 bugs that the `WHERE` oracle cannot find. A threat to this is that the manual analysis was only a best effort comparison.

Limitations. Our testing does not apply to transactions, window functions, sequences, and non-deterministic functions. Queries can have ambiguous results, which limits the technique; this affects subqueries in particular [Rigger and Su 2020a], which we did not test. We found that especially SQLite has some peculiarities, such as treating the integer `0` and floating-point number `0.0` as the same number. CockroachDB and TiDB are distributed DBMS, and we tested only their SQL components. We considered only the most commonly used aggregate functions, many of which were straightforward to decompose. An overview of various decompositioning strategies, also for other classes of aggregate functions, is given by Jesus et al. [2015].

Implementation order. Developers might wonder in which order to implement test oracles. The `WHERE` oracle is the simplest, but most effective oracle to implement. Only when this oracle does not find any more bugs is it useful to implement the subsequent oracles, which generate additional clauses in addition to the **`WHERE`** clause. Generating the simplest test case possible is preferable, since it speeds up the triaging, reduction, and understanding of bugs. Similarly, the `HAVING` oracle should be implemented only after the `GROUP BY` oracle cannot find any additional bugs, since the `HAVING`

Listing 17. Query Partitioning could be applied to, for example, test the semantics of individual operators.

```
CREATE TABLE t0(c0 INT, c1 INT);
Ⓞ SELECT * FROM t0 WHERE c0 <=> c1;
Ⓟ SELECT * FROM t0 WHERE (c0 IS NULL and c1 IS NULL) UNION ALL
        SELECT * FROM t0 WHERE (c0 IS NOT NULL and c1 IS NOT NULL AND c0 = c1);
```

oracle also generates `GROUP BY` clauses. The aggregate test oracles are more complex and specialized to an individual aggregate function; thus, we believe that these could be implemented last.

Other partitioning strategies. Besides Ternary Query Partitioning, a number of additional partitioning strategies are imaginable. As one example, the partitioning could be specific to operators or functions. For example, MySQL provides an operator <=>, which is similar to the equality operator, but evaluates also to a boolean value when comparing to a `NULL` value. A query using the operator in a predicate could be partitioned by replacing it with a series of `IS NULL` checks and an equality comparison. Listing 17 gives a concrete example of an original query Ⓞ, which is translated to two partitioning queries Ⓟ that are expected to compute the same result.

7 RELATED WORK

The closest related work is Pivoted Query Synthesis (PQS) and Nonoptimizing Reference Engine Construction (NoREC), which both aim to find logic bugs and were both extensively discussed. A number of approaches to test various aspects of DBMS and related software have been proposed.

Differential testing of DBMS. Differential testing [McKeeman 1998] refers to a testing technique where a single input is passed to multiple systems that are expected to produce the same output; if the systems disagree on the output, a bug in at least one of the systems has been detected. It has proven to be effective in many domains [Brummayer and Biere 2009; Kapus and Cadar 2017; McKeeman 1998; Yang et al. 2011]. Slutz [1998] applied this technique for testing DBMS in a system called RAGS by generating SQL queries that are sent to multiple DBMS and then observing differences in the output sets. While the approach was effective, the author stated that the small common core and the differences between different DBMS were a challenge, which was also noted by Rigger and Su [2020a,b]. Differential testing was, however, found to be useful to compare query plans within a DBMS, or the performance of multiple versions of a DBMS. Specifically, Gu et al. [2012] used options and hints to force the generation of different query plans, to then rank the accuracy of the optimizer based on the estimated cost for each plan. Jung et al. [2019] used differential testing in a system called *APOLLO* to find performance regression bugs in DBMS, by executing a SQL query on an old and newer version of a DBMS.

Solver-based testing of DBMS. ADUSA is a query-aware database generator that generates inputs as well as the expected result for a query [Khalek et al. 2008]. It translates the schema and query to an Alloy specification, which is subsequently solved. The approach could reproduce various known and injected bugs in MySQL, HSQLDB, and also find a new bug in Oracle Database. We believe that the high overhead that solver-based approaches incur might inhibit such approaches from finding more DBMS bugs.

Random and targeted queries. Many query generators have been proposed for purposes such as bug-finding and benchmarking. SQLsmith is a widely-used, open-source random query generator, which has found over 100 bugs in widely-used DBMS [Seltenreich 2019]. Bati et al. proposed an approach based on genetic algorithms to incorporate execution feedback for generating queries [Bati et al. 2007]. SQLFUZZ [Jung et al. 2019] also utilizes execution feedback and randomly generates queries using only features that are supported by all the DBMS systems they considered. Khalek et al. proposed generating both syntactically and semantically valid queries based on a solver-backed approach [Abdul Khalek and Khurshid 2010]. All these random-query generators can be used to find bugs such as crashes and hangs in DBMS. When paired with the test oracles proposed in this paper, they could also be used to find logic bugs.

Random and targeted databases. Many approaches have been proposed to automatically generate databases. Given a query and a set of constraints, QAGen [Binnig et al. 2007b; Lo et al. 2010] generates a database that matches the desired query results by combining traditional query processing and symbolic execution. Reverse Query Processing takes a query and a desired result set as an input to then generate a database that could have produced the result set [Binnig et al. 2007a]. As discussed above, ADUSA is a query-aware database generator [Khalek et al. 2008]. Gray et al. discussed a set of techniques utilizing parallel algorithms to quickly generate billions-record databases [Gray et al. 1994]. DGL is a domain-specific language that generates input data following various distributions and inter-table correlations based on *iterators* that can be composed [Bruno and Chaudhuri 2005]. An improved database generation might enable TLP to find additional bugs.

Metamorphic testing. Metamorphic testing [Chen et al. 1998] addresses the test oracle problem by, based on an input and output of a system, generating a new input for which the result is known. Central in this approach is the metamorphic relation, which can be used to infer the expected result. This technique has been applied successfully in various domains [Chen et al. 2018; Donaldson et al. 2017; He et al. 2020; Le et al. 2014; Segura and Zhou 2018; Winterer et al. 2020]. The test oracle proposed as part of this paper is a metamorphic one, since based on the original query and its result set, we generate partitioning queries, whose composed result sets must be equal to the original query's result set. Note that also NoREC is a metamorphic test oracle.

Optimizing aggregate functions. We adopted ideas from optimizing aggregate functions to testing them. Cohen [2006] studied user-defined aggregate functions in the context of query optimization, query rewriting, and view maintenance. Yu et al. [2009] studied the interfaces and implementation of user-defined aggregate functions in the context of distributed aggregation. Since we decompose a query to partitioning queries, which can be computed independently, we study a similar problem. However, in contrast to their work, our goal is not to decompose the query for optimization, but to test the DBMS. [Jesus et al. 2015] surveyed the techniques in distributed data aggregation, provided a formal framework, and characterized the types of aggregate functions.

8 CONCLUSION

This paper has presented the general idea of Query Partitioning, and a concrete instantiation of this idea, termed Ternary Logic Partitioning (TLP). The core idea of Query Partitioning is to partition a query into multiple so-called partitioning queries, each of which computes a partition of the result. By using a composition operator, the partitions can be combined to yield the same result as the original query; if the result differs, a bug in the DBMS has been detected. TLP partitions queries based on a boolean predicate, which can either evaluate to `TRUE`, `FALSE`, or `NULL`. TLP can detect bugs in various features, including `WHERE` clauses, `GROUP BY` clauses, `HAVING` clauses, `DISTINCT` queries, and aggregate functions. Our evaluation on six widely-used DBMS has demonstrated that TLP is highly effective and general, as it could detect 77 logic bugs, at least 17 of which cannot be detected by existing techniques. Despite TLP's effectiveness, we believe that a number of additional query partitioning strategies can be devised, which might allow finding additional bugs in DBMS.

REFERENCES

Shadi Abdul Khalek and Sarfraz Khurshid. 2010. Automated SQL Query Generation for Systematic Testing of Database Engines. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. ACM, New York, NY, USA, 329–332. https://doi.org/10.1145/1858996.1859063

M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. 1976. System R: Relational Approach to Database Management. *ACM Trans. Database Syst.* 1, 2 (June 1976), 97–137. https://doi.org/10.1145/320455.320457

Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. 2007. A Genetic Approach for Random Testing of Database Systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (Vienna, Austria) *(VLDB '07)*. VLDB Endowment, 1243–1251.

Carsten Binnig, Donald Kossmann, and Eric Lo. 2007a. Reverse Query Processing. *Proceedings - International Conference on Data Engineering*, 506–515. https://doi.org/10.1109/ICDE.2007.367896

Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. 2007b. QAGen: Generating Query-Aware Test Databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (Beijing, China) *(SIGMOD '07)*. Association for Computing Machinery, New York, NY, USA, 341–352. https://doi.org/10.1145/1247480.1247520

Robert Brummayer and Armin Biere. 2009. Fuzzing and Delta-Debugging SMT Solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories* (Montreal, Canada) *(SMT '09)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/1670412.1670413

Nicolas Bruno and Surajit Chaudhuri. 2005. Flexible Database Generators. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 1097–1107.

Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. 2006. Generating Queries with Cardinality Constraints for DBMS Testing. *IEEE Trans. on Knowl. and Data Eng.* 18, 12 (Dec. 2006), 1721–1725. https://doi.org/10.1109/TKDE.2006.190

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* (Seattle, Washington, USA) *(PODS '98)*. Association for Computing Machinery, New York, NY, USA, 34–43. https://doi.org/10.1145/275487.275492

Tsong Y Chen, Shing C Cheung, and Shiu 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 ....

Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1, Article Article 4 (Jan. 2018), 27 pages. https://doi.org/10.1145/3143561

Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming Compiler Fuzzers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 197–208. https://doi.org/10.1145/2491956.2462173

Cockroach Labs. 2020. Cockroach Labs, the company building CockroachDB. https://www.cockroachlabs.com/

E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (June 1970), 377–387. https://doi.org/10.1145/362384.362685

Sara Cohen. 2006. User-Defined Aggregate Functions: Bridging Theory and Practice. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/1142473.1142480

Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2012. Understanding Integer Overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 760–770.

Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 93 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133917

Leo Giakoumakis and César A. Galindo-Legaria. 2008. Testing SQL Server's Query Optimizer: Challenges, Techniques and Experiences. *IEEE Data Eng. Bull.* 31 (2008), 36–43.

Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (June 1993), 73–169. https://doi.org/10.1145/152610.152611

Goetz Graefe. 2011. Modern B-Tree Techniques. *Found. Trends Databases* 3, 4 (April 2011), 203–402. https://doi.org/10.1561/1900000028

Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. *SIGMOD Rec.* 23, 2 (May 1994), 243–252. https://doi.org/10.1145/191843.191886

Zhongxian Gu, Mohamed A. Soliman, and Florian M. Waas. 2012. Testing the Accuracy of Query Optimizers. In *Proceedings of the Fifth International Workshop on Testing Database Systems* (Scottsdale, Arizona) *(DBTest '12)*. ACM, New York, NY, USA, Article 11, 6 pages. https://doi.org/10.1145/2304510.2304525

Paolo Guagliardo and Leonid Libkin. 2017. A Formal Semantics of SQL Queries, Its Validation, and Applications. *Proc. VLDB Endow.* 11, 1 (Sept. 2017), 27–39. https://doi.org/10.14778/3151113.3151116

L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. 1989. Extensible Query Processing in Starburst. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data* (Portland, Oregon, USA) *(SIGMOD '89)*. Association for Computing Machinery, New York, NY, USA, 377–388. https://doi.org/10.1145/67544.66962

Pinjia He, Clara Meister, and Zhendong Su. 2020. Structure-Invariant Testing for Machine Translation. In *Proc. of the 42nd International Conference on Software Engineering (ICSE)*.

Kenneth Houkjær, Kristian Torp, and Rico Wind. 2006. Simple and Realistic Data Generation. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 1243–1246.

P. Jesus, C. Baquero, and P. S. Almeida. 2015. A Survey of Distributed Data Aggregation Algorithms. *IEEE Communications Surveys Tutorials* 17, 1 (2015), 381–404.

Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. *Proc. VLDB Endow.* 13, 1 (Sept. 2019), 57–70. https://doi.org/10.14778/3357377.3357382

Timotej Kapus and Cristian Cadar. 2017. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 590–600.

S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid. 2008. Query-Aware Test Generation Using a Relational Constraint Solver. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. IEEE Computer Society, Washington, DC, USA, 238–247. https://doi.org/10.1109/ASE.2008.34

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334

Eric Lo, Carsten Binnig, Donald Kossmann, M. Tamer Özsu, and Wing-Kai Hon. 2010. A framework for testing DBMS features. *The VLDB Journal* 19, 2 (01 Apr 2010), 203–230. https://doi.org/10.1007/s00778-009-0157-y

Michaël Marcozzi, Qiyi Tang, Alastair F. Donaldson, and Cristian Cadar. 2019. Compiler Fuzzing: How Much Does It Matter? *Proc. ACM Program. Lang.* 3, OOPSLA, Article 155 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360581

William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. 2008. Generating Targeted Queries for Database Testing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. ACM, New York, NY, USA, 499–510. https://doi.org/10.1145/1376616.1376668

MySQL. 2020. MySQL Homepage. https://www.mysql.com/

Andrea Neufeld, Guido Moerkotte, and Peter C. Lockemann. 1993. Generating Consistent Test Data: Restricting the Search Space by a Generator Formula. *The VLDB Journal* 2, 2 (April 1993), 173–214.

Andrew Pavlo and Matthew Aslett. 2016. What's Really New with NewSQL? *SIGMOD Rec.* 45, 2 (Sept. 2016), 45–55. https://doi.org/10.1145/3003665.3003674

PingCAP. 2020. TiDB | SQL at Scale. https://pingcap.com

Meikel Poess and John M. Stephens. 2004. Generating Thousand Benchmark Queries in Seconds. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 1045–1053.

PostgreSQL. 2020. PostgreSQL Homepage. https://www.postgresql.org/

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

John Regehr. 2010. A Guide to Undefined Behavior in C and C++. https://blog.regehr.org/archives/213

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. *SIGPLAN Not.* 47, 6 (June 2012), 335–346. https://doi.org/10.1145/2345156.2254104

Manuel Rigger and Zhendong Su. 2020a. Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction.

Manuel Rigger and Zhendong Su. 2020b. Testing Database Engines via Pivoted Query Synthesis. https://arxiv.org/abs/2001.04174

Sergio Segura and Zhi Quan Zhou. 2018. Metamorphic Testing 20 Years Later: A Hands-on Introduction. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 538–539. https://doi.org/10.1145/3183440.3183468

Andreas Seltenreich. 2019. SQLSmith. https://github.com/anse1/sqlsmith

Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. 2011. Dynamic Race Detection with LLVM Compiler. In *Proceedings of the Second International Conference on Runtime Verification* (San Francisco, CA) *(RV'11)*. Springer-Verlag, Berlin, Heidelberg, 110–114. https://doi.org/10.1007/978-3-642-29860-8_9

Donald R Slutz. 1998. Massive stochastic testing of SQL. In *VLDB*, Vol. 98. 618–622.

SQLite. 2020. SQLite Homepage. https://www.sqlite.org/

Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: Fast Detector of Uninitialized Memory Use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (San Francisco, California) *(CGO '15)*. IEEE Computer Society, USA, 46–55.

Manasi Vartak, Venkatesh Raghavan, and Elke A. Rundensteiner. 2010. QRelX: Generating Meaningful Queries That Provide Cardinality Assurance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 1215–1218. https://doi.org/10.1145/1807167.1807323

Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. In *Proceedings of the 41th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. 2009. Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 247–260. https://doi.org/10.1145/1629575.1629600