Data-Oriented Differential Testing of Object-Relational Mapping Systems

Thodoris Sotiropoulos
Athens University of Economics and Business
Athens, Greece
theosotr@aueb.gr

Stefanos Chaliasos
Athens University of Economics and Business
Athens, Greece
schaliasos@aueb.gr

Vaggelis Atlidakis
Columbia University
New York, USA
vatlidak@cs.columbia.edu

Dimitris Mitropoulos
Athens University of Economics and Business
National Infrastructures for Research and Technology - GRNET
Athens, Greeece
dimitro@aueb.gr

Diomidis Spinellis
Athens University of Economics and Business
Athens, Greece
dds@aueb.gr

***Abstract*—We introduce, what is to the best of our knowledge, the first approach for systematically testing Object-Relational Mapping (ORM) systems. Our approach leverages differential testing to establish a test oracle for ORM-specific bugs. Specifically, we first generate random relational database schemas, set up the respective databases, and then, we query these databases using the APIs of the ORM systems under test. To tackle the challenge that ORMs lack a common input language, we generate queries written in an abstract query language. These abstract queries are translated into concrete, executable ORM queries, which are ultimately used to differentially test the correctness of target implementations. The effectiveness of our method heavily relies on the data inserted to the underlying databases. Therefore, we employ a solver-based approach for producing targeted database records with respect to the constraints of the generated queries. We implement our approach as a tool, called CYNTHIA, which found 28 bugs in five popular ORM systems. The vast majority of these bugs are confirmed (25 / 28), more than half were fixed (20 / 28), and three were marked as release blockers by the corresponding developers.**

***Index Terms*—Object-Relational Mapping, Differential Testing, Automated Testing**

I. Introduction

Object-Relational Mapping (ORM) is an established programming technique [1], [2], [3] that has emerged as a solution to the *Object-Relational Impedance Mismatch* problem [4], [5]. ORM provides an object-oriented interface atop relational databases. Through that, the objects of a program can be easily saved and retrieved from the secondary storage without requiring boilerplate code for mapping application data to database records. ORM not only boosts developer productivity and reduces maintenance costs [4], [6], but also promotes portability because it abstracts away differences of Database Management Systems (DBMS) [4], [6].

Currently, there is a plethora of ORM implementations: through a simple Github search, one runs into more than 50 ORM frameworks, written for almost every language. Indicative examples include Django and SQLAlchemy for Python, Hibernate for Java, ActiveRecord for Ruby, and Sequelize for JavaScript. These systems are used by millions of applications [7] and are adopted by many popular organizations, such as Dropbox, Gitlab, and OpenStack [8], [9], [10].

Despite their wide industrial adoption, the automated testing of ORM systems is an overlooked problem. Current testing efforts mainly use manually-written test suites, which, as we demonstrate, are often insufficient for ensuring the correctness of ORM implementations. Yet, ORM implementations are complex [4] (typically consist of thousands lines of code) and, unfortunately, involve a high density of bugs. For example, the ORM implementation in the Django web framework is the component that suffers from the most bugs [11]: 23% of the reported bugs in Django are related to the ORM component, and they are significantly more than the reported bugs associated with the secondly affected component (8%). Such ORM bugs lead to incorrect interactions with the underlying database and cause frustrating crashes [12], wrong store and retrieval of data [13], and even security vulnerabilities [14], [15].

To detect bugs in ORM implementations, we propose a differential testing approach. At a high-level, our approach exercises ORM systems by constructing equivalent queries written in the target ORM implementations, and then compares query results for mismatches. We begin by generating a random database schema used to set up databases across multiple DBMSs. We test the functionality of ORMs by querying the databases using each ORM's API. However, since ORM systems do not share a common input format, we design an abstract query language which is close to ORM APIs. This allows us to build expressive queries that exercise diverse functionality combinations across ORM implementations. Finally, we use ORM-specific *translators* to convert abstract queries into concrete ones, which are executable in the corresponding ORM implementations.

Our differential testing approach is *data-oriented*: beyond queries, it is the data inserted to the underlying databases that affect the effectiveness of the testing efforts. We em-

```
1  from django.db import models
2  class Person(models.Model):
3    age = models.IntegerField()
4    name = models.CharField(max_length=20)
5  ...
6  p1 = Person(age=31, name="John")
7  p1.save()
8  p2 = Person.objects.get(age=32)
9  p2.delete()
```

Fig. 1: Example CRUD operations using the Django ORM.

ploy a solver-based approach for generating *targeted* database records with respect to the constraints of the generated abstract queries. This improves the effectiveness of differential testing because it minimizes the number of queries where ORMs return empty results. Our approach goes beyond the existing body of work in compiler and programming language testing [16], [17], [18], [19], [20], and addresses several challenges specific to ORM systems, such as lack of a common input, data generation, database schema generation, or DBMS setup. Specifically, we make the following contributions:

- We introduce the first automatic, data-oriented differential testing approach for ORM system implementations.
- We implement CYNTHIA, an extensible open-source framework for systematically testing well-established ORM implementations.
- We provide experimental evidence showing that our solver-based approach is an effective technique to generate data that are useful for differential testing.
- We use CYNTHIA to test five popular ORM systems on four widely-used database engines, and find 28 unique bugs. The vast majority of these bugs are confirmed (25 / 28), more than half were fixed (20 / 28), and three were marked as release blockers by the corresponding developers.

Availability. Our system, CYNTHIA, is available as open-source software under the GNU General Public License v3.0 at https://github.com/theosotr/cynthia. The research artifact is available at https://doi.org/10.5281/zenodo.4455486.

II. Background & Motivation

We provide a brief overview of object-relational mapping and an illustrative example of bugs that our approach can detect in the related tools and frameworks. Then, we briefly explain why we adopt differential testing for detecting these bugs. Finally we enumerate the main challenges associated with differential testing of object-relational mapping systems.

A. Object-Relational Mapping Systems

Object-Relational Mapping provides an abstraction over relational data that enables programmers to interact with their databases through the object-oriented programming paradigm. In this context, a database schema (tables and their inter-relationships) is abstracted through classes, called *models*, and the associated database records are represented via objects of these classes. ORM systems then provide a rich API for basic Create, Read, Update, and Delete (CRUD) operations on database records as well as more advanced features, such as transaction management or query caching.

```
1  q1 = T1.objects.using("mysql")
2  q2 = T2.objects.using("mysql")
3  q3 = T3.objects.using("mysql")
4  //ProgrammingError: "You have an error in your
       SQL syntax"
5  q1.union(q2).union(q3)
6  // Generated SQL
7  (SELECT `t1`.`id` FROM `t1`)
8  UNION (
9    (SELECT `t2`.`id` FROM `t2`)
10   UNION
11   (SELECT `t3`.`id` FROM `t3`))
```

Fig. 2: Django generates MySQL query with invalid syntax.

Figure 1 shows an example of database interactions using the Django ORM system [9]. The code first declares a class that maps to a table and to its associated columns in the underlying database (lines 2–4). Using this class, the code then runs simple queries. Specifically, the code creates a class object (line 6), and based on this object, creates a new database record by calling the `save()` method (line 7). Then, the code fetches a single record from the database matching certain criteria (line 8), and then deletes this record (line 9).

ORM system APIs provide a higher level of abstraction that hides the mechanics of SQL queries from the programmer. For example, the `save()` method results in an SQL `INSERT` statement, which remains transparent to the programmer.

B. Bugs in Object-Relational Mapping Systems

To motivate the design of our testing approach, we discuss two indicative bugs found in well-established ORM systems.

Bug in Django. Consider the Django query shown in Figure 2 (lines 1–5). This query first fetches the records of tables `t1`, `t2`, and `t3` (lines 1–3), and it then performs a chain of unions (line 5) in order to combine the results of the individual queries. When we run this Django code on MySQL (version 8.0.4), Django produces and runs the SQL query shown on lines 7–11. This SQL query is invalid on MySQL and the Django program crashes with a *django.db.utils.ProgrammingError: (1064, "Error in SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'UNION'")*. This bug was detected by our approach, and was confirmed by the Django developers.

When the Django code shown in Figure 2 is run on another DBMS, such as SQLite or PostgreSQL, Django produces a valid SQL query. Such inconsistencies indicate that ORM bugs may appear (or not) depending on the underlying DBMS. Although DBMSs share common functionality, they differ significantly from each other [21]. Therefore, an ORM needs to abstract away such differences and take care of running the same ORM code on different DBMSs reliably. Unfortunately, this complicates the design of ORMs: bugs may occur when an ORM fails to produce a valid SQL query with respect to a certain DBMS.

Bug in peewee. Figure 3 shows another ORM bug detected by our approach. On lines 1–3, the code creates a simple query using the peewee ORM. The query defines a simple

```
1  expr = (1 + T.col)
2  squared = (expr * expr)
3  T.select(fn.sum(expr), fn.avg(squared)).all()
4  // Generated SQL
5  SELECT SUM(1 + "t"."col"),
6         AVG(1 + "t"."col" * 1 + "t"."col")
7  FROM "t" AS "t"
```

Fig. 3: Logic error detected in peewee ORM.

expression `expr` given by the addition of a table's column with 1 (line 1). The code then forms a simple query that applies the function `SUM` to `expr`, and `AVG` to the square of `expr` (lines 2, 3). The peewee ORM translates this high-level query into the *incorrect* SQL query shown on lines 5–7. In this SQL query, the expression passed to the aggregate function `AVG` is not in the expected format because the sub-expressions are not wrapped in parentheses: peewee incorrectly produces $\texttt{AVG}(1 + col * 1 + col)$ instead of $\texttt{AVG}((1 + col) * (1 + col))$. This bug was confirmed and fixed by the peewee developers immediately after our report.

Unlike the Django bug discussed earlier, this peewee bug is more subtle: Although peewee generates a grammatically and semantically valid SQL query, this query produces incorrect results. Unlike crashes, such subtle bugs cannot be detected through a naive fuzzing approach. This explains our primary design choice to adopt differential testing.

C. Differential Testing of ORM Systems

To find bugs similar to the ones discussed above, we need to systematically determine whether the SQL query generated by an ORM system is correct or not. To do so, we need to define a test oracle. Nevertheless, establishing a test oracle for ORM-specific bugs is not straightforward. For example, we are unable to decide whether the SQL query generated by Django (Figure 2) is incorrect, unless we have domain knowledge that nested unions are indeed supported by MySQL, and therefore, it is a bug from Django which failed to produce a grammatically correct SQL query involving nested unions. Worse, there is no an easy way to tell that the peewee bug of Figure 3 is buggy. Although this query runs successfully on all DBMSs, we cannot be sure that this query indeed fetches the expected results from the database.

To address the test oracle problem, we employ *differential testing*[22], a generally-applicable method for testing equivalent implementations. Differential testing provides us with an oracle as follows. We feed the same test input (e.g., query) to two equivalent implementations (e.g., Django and peewee), and then compare their results. A mismatch found in the results of the implementations under test indicates a potential bug in at least one of them. For example, through differential testing, we run the query associated with nested unions (Figure 2) on MySQL, this time using the API of peewee. Peewee executes the given query on MySQL without errors. This helps us to identify that there is a bug in Django implementation. Similarly, for the peewee query shown in Figure 3, we construct its counterpart written in Django, only to see that Django and peewee produce different results.

D. Challenges

Our approach is inspired by prior work on compiler and programming language testing [16], [17], [18], [19], [20], a domain where differential testing has been successfully used in the past. However, applying differential testing on ORM systems is not straightforward, and it involves several new challenges.

Challenge 1: Lack of a common specification and input language. ORM systems do not implement a common specification or standard. Therefore, differences in ORM results may be due to valid but inconsistent implementations and not due to actual bugs. Furthermore, each ORM offers its own APIs and, to make matters worse, these APIs may even be exposed through different programming languages. As a result, differential testing cannot be uniformly applied to test ORM systems in a straightforward manner.

Challenge 2: Non-deterministic query results. In some ORM systems, it is possible to write a query that leads to an SQL statement that produces a non-deterministic result, i.e., the result depends on the implementation of the underlying DBMS. An example of such query is when the results are not ordered. In this case, the DBMS is free to return results in any order. Another example is when the resulting SQL query has a column a and an aggregate function in the `SELECT` part, but the query does not define a `GROUP BY` clause on the column a. According to the SQL standard, selecting a column and an aggregate function, without specifying a `GROUP BY` clause leads to an *ambiguous* query whose results are not deterministic. To compare the results of the ORM systems under test in a meaningful way, we have to deal with this non-determinism.

Challenge 3: DBMS-dependent results. As shown previously (Figure 2), there are ORM bugs that are DBMS-specific, i.e., the bugs are triggered only when the ORM code works on a certain DBMS. To effectively capture such bugs, we need to differentially test the ORM systems on multiple DBMSs. At the same time, though, differences between the underlying DBMSs (e.g., two DBMSs may have different semantics on arithmetic expressions) must not affect the comparisons of ORMs. Finally, for performing safe comparisons, the ORM code needs to run on a common reference, i.e., the ORM queries need to run on the same database.

Challenge 4: Data generation. Beyond ORM queries, we have to generate appropriate data to populate the databases so that ORM systems produce *non-trivial* results in response to given queries. In this way, we can reveal logic errors that cause ORMs to fetch the wrong data from the database. For example, it is impossible to detect the peewee bug of Figure 3 when the underlying database contains no records.

III. Testing Approach

Our approach for testing ORMs is automated as shown in Figure 4. It takes as input the ORM systems to test, and the DBMSs where the ORM queries will run. *Schema Generation* is an initial phase where we generate a number of relational database schemas. During the *Setup* phase, we build

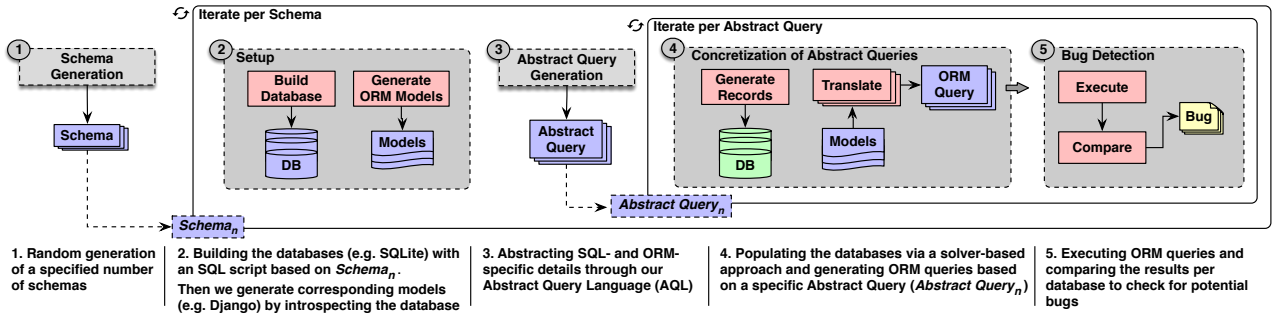

Fig. 4: Overview of our approach for automatically testing ORMs.

the different databases—one for each provided DBMS—with respect to the schema generated during the first step. Then, we proceed to the *Abstract Query Generation* phase which involves the generation of queries written in the Abstract Query Language (AQL). We design this language to abstract ORM- and SQL-specific details and provide a common reference for testing ORMs, thus addressing "Challenge 1". By design, AQL queries never lead to ambiguous ORM queries ("Challenge 2"). However, AQL queries may be unordered. In this case, we interpret query results as a set of rows rather than a sequence (see also Section III-D). In the *Concretization of Abstract Query* phase we use ORM-specific translators to translate each query into a concrete one. To deal with "Challenge 4" and minimize the number of cases where ORMs produce empty results, we synthesize database records using a solver-based approach. In the last step, i.e. *Bug Detection*, we execute the ORM queries on diverse DBMSs, and compare their results. A mismatch in the outputs indicates a potential bug in at least one ORM. Notably, testing the ORM code across different DBMSs enables us to find DBMS-dependent bugs ("Challenge 3").

A. Schema Generation & Setup

We generate a number of schemas that capture the structure of the databases on which each ORM under test operates. Each schema s is a collection of tables and their associated columns. Each column has a type that can be a **serial** (primary key of the table), a number (i.e., **integer** or **real**), a string, or **foreign** t which indicates a table's relationship with another table t of the schema. We omit schema details such as indexes, views or column constraints (e.g., unique), as these constructs do not affect the querying and translation mechanisms of ORMs, and therefore, are beyond the scope of this paper.

Our method randomly generates a user-defined number of schemas. For each table, the schema generation algorithm creates a **serial** column named "id" that stands for the primary key of the table, to guarantee that each record in the table is unique and that there is no ambiguity in the data inserted into the table. The remaining fields of the table are randomly generated (optionally based on a deterministic procedure).

We use the schemas generated in the previous step to set up and instantiate the respective DBMSs and ORMs. To set up DBMSs, we automatically construct an SQL script containing all `CREATE TABLE` statements for creating the tables defined in a provided schema along with their columns. Then, we automatically generate the models for each ORM under test by examining the structure of the newly-created databases. To this end, we leverage tools used to ease ORM porting to existing databases. These tools make a connection to an existing database, introspect its structure, and automatically construct the respective ORM model classes. An example of such tool is the command `manage.py inspectdb` found in the Django project [23].

$$
\begin{aligned}
q \in \mathit{Query} &::= \textbf{eval}\ qs \mid qs[i] \mid qs[i:i] \mid \textbf{fold}\ \{\ (l : \alpha\ e)^{+}\ \}\ qs \\
qs \in \mathit{QuerySet} &::= \textbf{new}\ t \mid \textbf{apply}\ \lambda\ qs \mid qs \cup qs \\
&\quad \mid qs \cap qs \\
\lambda \in \mathit{Func} &::= \textbf{filter}\ p \mid \textbf{map}\ d \mid \textbf{unique}\ \phi \\
&\quad \mid \textbf{sort}\ (\phi\ \textbf{asc}) \mid \textbf{sort}\ (\phi\ \textbf{desc}) \\
d \in \mathit{FieldDecl} &::= l : e \mid \textbf{hidden}\ l : e \mid d; d \\
p \in \mathit{Pred} &::= \phi \oplus e \mid p \wedge p \mid p \vee p \mid \neg p \\
e \in \mathit{Expr} &::= c \mid \phi \mid \alpha\ e \mid e + e \mid e - e \mid e * e \mid e/e \\
\phi \in \mathit{Field} &::= t.c \mid l \mid \phi.c \\
\alpha \in \mathit{AggrFunc} &::= \textbf{count} \mid \textbf{sum} \mid \textbf{avg} \mid \textbf{max} \mid \textbf{min} \\
\oplus \in \mathit{BinaryOp} &::= = \mid > \mid \geq \mid < \mid \leq \\
&\quad \mid \textbf{contains} \mid \textbf{startswith} \mid \textbf{endswith}
\end{aligned}
$$

Fig. 5: The syntax of the Abstract Query Language (AQL).

B. Abstract Query Generation

Following the *Schema Generation & Setup* phases, we start a testing session for each individual schema. A testing session involves the generation of multiple valid queries (with respect to the provided schema) that are likely to reveal bugs in the ORMs under test. These queries are represented in the *Abstract Query Language (AQL)*, which is close to the APIs and the functionality of ORMs, and provides a wide range of operations (through a functional notation) that are commonly supported by the querying mechanism of ORMs. AQL operations include filtering, sorting, aggregate functions, creation of compound expressions, field aliasing, or union and intersection of queries. By contrast, raw SQL dialect is too low-level and many ORMs are not aware of SQL constructs. Also, the SQL language is not rich enough to express and capture the different API calls of ORMs. For example, the same SQL query can be produced by calling different combinations of ORM's API methods. Since our focus is on detecting bugs in ORMs by exercising different combinations of their API calls, we design AQL.

1) Abstract Query Language: Figure 5 shows the syntax of AQL. A query in AQL is the evaluation of a query set (**eval** qs).

```
1  apply (filter "addCol" > 5
2    apply (map "addCol": t1.colA + t1.t2.colB
3      new t1))
4
5  SELECT t2.colA + t2.colB AS "addCol"
6  FROM t1 as "t1"
7  JOIN t2 AS "t2" ON (t1.t2_id = t2.id)
8  WHERE (t2.colA + t2.colB > 5)
```

Fig. 6: Example AQL query and its equivalent SQL query.

Conceptually, a query set evaluates to a set or to a sequence of records (in case the query set is ordered). Operations such as indexing or slicing, can be applied to the result of a query set, while AQL also supports folding. The function **fold** aggregates the result of a query set into labeled scalar values by applying one or more aggregate functions.

The simplest form of a query set is **new** t, which creates a new query set from the specified table t. When this query is evaluated, it returns all records of the table t. Then, various operations can be applied to a query set through the **apply** construct. In particular, AQL provides the **filter** p function that returns all records of the query set that satisfy the given predicate p. The **map** function is used to create new compound fields using existing fields found in the given query set. Specifically, **map** expects a sequence of field declarations of the form $l : e$. This declaration creates a new field in the current query set by binding the expression e to the label l. Optionally, a field can be marked as hidden meaning that it is not part of the query set, but it is used for creating other fields (hidden fields are similar to temporary variables). The function **sort**, sorts the provided query set according to the field ϕ in an ascending or a descending order, while the **unique** primitive removes duplicate records with respect to the provided field ϕ. Finally, AQL supports the combination of two query sets through the union and intersection operations.

A predicate consists of comparison operators (i.e., $\phi \oplus e$) which are used to compare the value of a field ϕ with the result of an expression e. A predicate may also contain the usual logical operators. An expression can be a constant c, a field reference ϕ, an application of an aggregate function, or an expression derived from the usual arithmetic operators. Finally, a field $\phi \in$ *Field* may be a reference to a column of a table, i.e., $t.c$, a label l created by the **map** function, or a reference to a column of a table's relationship (e.g., $t_1.t_2.c$).

Figure 6 shows an example query written in AQL and its equivalent query written in SQL. In this AQL query, we apply two functions. First, we apply **map** to the query set given by `new t1` (lines 2–3) in order to create a new field named "addCol" given by the addition between the `t1.colA` and `t1.t2.colB` columns. Notice that since the latter column refers to a column of the table `t2`, which has a relationship with the original table `t1`, in SQL this is interpreted as a `JOIN` between `t1` and `t2` (line 7). Finally, we apply `filter` to get the records satisfying `addCol > 5` (line 1).

Remark. AQL currently supports only read operations. The implementations of ORM API methods associated with read operations are much more complex than those related to write operations (a write operation is straightforwardly translated into `INSERT`, `DELETE` or `UPDATE` queries). Thus, examining read operations for finding bugs is more promising. Note though that AQL can be easily extended for supporting write queries. Also, supporting write operations would not require to take into account schema properties that we are currently ignoring (e.g., column constraints), because such properties affect the configuration of ORM models and not the way an ORM translates a write query into an SQL statement.

Algorithm 1: Generating Abstract Queries

```
1  fun genQuerySet(σ, min, max) =
2      stopCond ← σ[depth] > min ∧ (σ[depth] > max ∨
       randBool())
3      if stopCond then σ[qs]
4      else
5          match chooseFrom(σ[qsNodes]) with
6              case NewNode ⇒
7                  t ←chooseTable(σ[schema])
8                  σ2 ← σ[qs → New(t), t → t]
9                  genQuerySet(σ2++, min, max)
10             case FilterNode ⇒
11                 p ←genPred(σ++, min, max)
12                 σ2 ← σ[qs → Apply(filter, p, σ[qs])]
13                 genQuerySet(σ2++, min, max)
14             case ... ⇒
15
16 fun genPred(σ, min, max) =
17     match chooseFrom(σ[predNodes]) with
18         case EqPredNode ⇒
19             f ←choosseField(σ)
20             Eq(f, genExpr(σ++, min, max))
21         case ... ⇒
22
23 fun genExpr(σ, min, max) =
24     match chooseFrom(exprNodes) with
25         case FieldRefNode ⇒
26             Field(chooseField(σ))
27         case ... ⇒
```

2) Generating AQL Queries: Algorithm 1 shows how we generate random AQL queries. Our algorithm generates queries that exercise all of the features supported by AQL, as well as different combinations of them. The main component of Algorithm 1 is the `genQuerySet` function (lines 1–14). This function generates an AQL query set by recursively constructing a valid AST node based on the syntax of Figure 5. The algorithm ensures that the depth of the resulting query set ranges within specific limits specified by the user-provided parameters *min* and *max* (see *stopCond*, line 2). The parameter σ keeps track of the state of the query set that is being generated. The initial state contains the schema ($\sigma[schema]$) based on which the algorithm creates table and column references. For what follows, the operation σ_{++} results in a new state where the value of $\sigma[depth]$ is incremented.

Our algorithm first constructs a new query set (**new** t) that queries a certain table (lines 6–9). To do so, we randomly choose a table to query from the underlying schema (line 7). Then, the algorithm updates the state σ in order to properly build the next available AST node in the next iteration. In particular, it initializes the AST of the current query set to

$$
\begin{aligned}
P(c, i) &\rightarrow c \\
P(f, i) &\rightarrow f_i \\
P(e_1 \oplus e_2, i) &\rightarrow P(e_1, i) \oplus P(e_2, i) \\
A(c, g) &\rightarrow c \\
A(f, g) &\rightarrow P(f, i) \quad i \in g \\
A(\textbf{count}\ e, g) &\rightarrow \mathit{len}(g) \\
A(\textbf{sum}\ e, g) &\rightarrow \sum_{i \in g} P(e, i) \\
A(\textbf{avg}\ e, g) &\rightarrow (\sum_{i \in g} P(e, i))/\mathit{len}(g) \\
A(\textbf{max}\ e, g) &\rightarrow \mathit{max}(e, g) \\
A(\textbf{min}\ e, g) &\rightarrow \mathit{min}(e, g) \\
A(e_1 \oplus e_2, g) &\rightarrow A(e_1, g) \oplus A(e_2, g)
\end{aligned}
$$

$$
\mathit{max}(e, g) = \begin{cases} P(e, i) & g = \{i\} \\ \mathit{ite}(P(e, i) > P(e, j), P(e, i), P(e, j)) & g = \{i, j\} \\ \mathit{ite}(P(e, i) > \mathit{max}(e, g'), P(e, i), \mathit{max}(e, g')) & g = i \cdot g' \end{cases}
$$

$$
\mathit{min}(e, g) = \ldots
$$

Fig. 7: Translating AQL expressions into SMT formulae.

New(t), while it sets the queried table to t (line 8). Then, it recursively calls `genQuerySet` to construct the next available AST nodes (line 9). For example, on lines 10–13, the algorithm applies **filter** to the current query set given by $\sigma[qs]$. To achieve this, the algorithm randomly generates a predicate p using the function `genPred` (lines 11, 16–21), and then extends the AST of the current query set to *Apply(filter* $p, \sigma[qs]$*)* (line 12). The AQL predicates and expressions are generated in a similar manner (see lines 16–21, 23–27). Finally, after producing a valid query set qs, we randomly decide for any operations applied to qs, i.e., slicing, indexing, or folding.

C. Concretization of Abstract Queries

During this phase, our approach derives multiple, concrete ORM queries (one for each target ORM) using ORM-specific translators (Section III-C2). Before producing these queries, our method populates the underlying databases with targeted data in order to enable differential testing (Section III-C1).

1) Generating Database Records: We follow a solver-based approach for generating a small number of targeted database records that satisfy the constraints of a given AQL query. Specifically, we model an AQL query and its constraints into *Satisfiability Modulo Theories (SMT)* formulae which we pass to a theorem prover. The theorem prover then solves the given SMT formulae and generates assignments that stand for the records inserted into the database. This approach improves the effectiveness of differential testing, as the corresponding ORMs will likely return non-empty results which in turn, can be used for detecting discrepancies in ORM outputs. In the following, we explain how we model an AQL query to SMT formulae.

Modeling table columns. We introduce a sequence of variables for every column of the queried table. Each variable in this sequence, namely x_i, represents the value of the column x in the i^{th} record of the table, where $1 \leq i \leq n$, and n is a specified number of records inserted into the database. After declaring these variables, we model the uniqueness of the table's id. To this end, we introduce the following constraint: $id_i \neq id_j$ for $1 \leq i \leq n$, where id_i refers to the id of the i^{th} record. Now, for what follows, $F(t_1, t_2)_i$ is the value of the foreign key defined in t_1 and refers to the table t_2, in the i^{th} record of t_1, while $V(t)$ gives the set of columns defined in table t, except for its id column.

Modeling joins. An AQL query may refer to columns defined in tables joined with the initial one. We traverse the AST of the given AQL query to identify such column references and compute the set of joins. For example, when encountering the $t_1.t_2.c$ reference, we know that there is join from table t_1 to t_2. After computing the set of joins, we introduce new variables for the columns of every joined table as we did for the root table. Then, for a join between two tables t_1, t_2, we create the following constraints, for $1 \leq i < j \leq n$:

- $F(t_1, t_2)_i = id(t_2)_i$
- $F(t_1, t_2)_i = F(t_1, t_2)_j \Rightarrow \bigwedge_{v \in V(t_2)} v_i = v_j$

The first constraint indicates that the foreign key of the source table t_1 must be the same with the id of the target table t_2 for all the records of t_1. The second constraint denotes that when there are two records in t_1, namely i and j, where the values of the foreign keys for t_2 are equal, all column values of the joined table t_2 must be also equal in the respective rows (e.g., $v_i = v_j$ for $v \in V(t_2)$). The last constraint ensures that two records of t_2 with the same id are identical.

Modeling AQL predicates. We model AQL predicates using two different ways, depending on whether the given predicate contains expressions consisting of an aggregate function (e.g., **sum**) or not. The simplest case is when a predicate does not contain an aggregate function. Such a predicate operates on all the records of the table. Converting a non-aggregate predicate is straightforward. For example, we convert the AQL equality predicate $t.c = e$ into:

$$\exists i.\ t.c_i = P(e, i) \text{ for } 1 \leq i \leq n$$

In the above formula, $t.c_i$ is the SMT variable that represents the value of the column $t.c$ in the i^{th} record of the table, while the function $P(e, i)$ encodes the given AQL expression e into a logical formula as shown in Figure 7. The above logical formula encodes the constraint that there must be at least one record in the table where the value of the column $t.c$ is equal with the value of the expression e.

An AQL predicate containing an aggregate function works on aggregated data formed by groups of records, and is conceptually similar to a condition that appears in the `HAVING` clause of an SQL query. To model such predicates as logical formulae, we first create a set G, consisting of a specified number of groups of records. Each group $g \in G$ includes all records that are identical based on a set of grouping fields GF. To compute the set of grouping fields GF, we traverse the AST of the given AQL query and add all column references that are not passed to an aggregate function. We then generate constraints so that the records of the same group are identical with respect to each field found in GF. Finally, we model aggregate predicates and their AQL expressions using the

```
1  import os, django
2  from django.db.models import *
3  os.environ.setdefault("DJANGO_SETTINGS_MODULE",
4                        "djangoproject.settings")
5  django.setup()
6  from project.models import *
7
8  addCol = F("colA") + F("t2__colB")
9  q = T1.objects.using("sqlite")\
10   .annotate(addCol=addCol).filter(addCol__gt=5)\
11   .values("addCol")
12 for r in q:
13   print("addCol", r["addCol"])
```

Fig. 8: The Django code related to the AQL query of Figure 6.

function $A(e, g)$ as defined in Figure 7. For example, the AQL predicate $t.a = \textbf{sum}\ t.b$ is translated into:

$$\exists g \in G.\ A(t.a, g) = A(\textbf{sum}\ t.b, g)$$

In the example above, $A(t.a, g)$ gives the SMT variable of the column $t.a$ associated with a *random* record of the group g. This is because $t.a$ is a grouping field (it is not part of an aggregate function) and all the records of g are the same with respect to the value of $t.a$. On the other hand, $A(\textbf{sum}\ t.b, g)$ aggregates all records of the group g based on the column $t.b$, i.e., $\sum_{i \in g} P(t.b, i)$. As Figure 7 indicates, the main difference between the functions $P(e, i)$ and $A(e, g)$ is that the former encodes the expression e as an SMT formula with regards to the record i, while the latter reasons about a group of records.

Modeling Unions & Intersections. Modeling unions and intersections is straightforward. Each sub-query of such an operation (e.g., $qs_1 \cup qs_2$) is translated into an SMT formula separately. Then the individual formulae are combined through logical operators. For unions, we use the disjunction operator ($\vee$), while we use $\wedge$ in case of intersection.

2) From Abstract Queries to Concrete ORM Queries: A translator takes an AQL query, converts it into an ORM query, and produces an executable file that runs the ORM query on a specified DBMS. Hence, a translator produces multiple executable files, one for each provided DBMS.

Every translator consists of three components. The first component adds the necessary boilerplate code for running the ORM query (e.g., imports, creating the connection with the database, etc.). The second component performs the translation. Specifically, it uses the API of the corresponding ORM to generate the actual ORM query. The last component dumps the results of the query to standard output, again by using the API of the specified ORM. When the query produces a sequence of records, the translator produces code that iterates over each element of the sequence and prints this element to standard output. To properly dump a record, the translator emits code that prints the value of every field defined in the AQL query. For example, when the query contains an application of **map**, the translator produces code that prints the value of every non-hidden field defined in **map**. When the given query does not apply **map**, then the id of the fetched records is printed. Finally, for queries returning scalar values (i.e., **fold**), the translator emits code that prints these scalar values.

Figure 8 shows the executable file that corresponds to the AQL query of Figure 6 and is produced by the Django translator. Notice that this file runs the Django query on SQLite. Lines 1–6 contain the necessary setup code for running the query, the actual Django query is on lines 8–11, while on lines 12–13, we print the results of the query.

D. Bug Detection

The last step of our testing approach is to run the executables produced by the translators and compare the output of these executables for mismatches. To do so, we run every executable and capture its standard output and standard error.

Our approach makes DBMS-specific comparisons: the output of a query q written in ORM o_1 and run on DBMS x is compared against the same query q written in another ORM o_2 and run on the *same* DBMS x. We do this because certain query features may be unsupported by some DBMS (e.g., MySQL does not support intersection queries.) Based on the above, our approach identifies mismatches and flags them as bugs, when one of the following conditions holds: (1) the same query written in two different ORMs produces different results on the same DBMS, or (2) a query written in a certain ORM runs successfully on a specific DBMS, but the same query written in another ORM fails on the same DBMS. The second condition allows us to detect cases where an ORM produced either a grammatically or semantically invalid SQL query with regards to a certain DBMS.

Remark. To make safe comparisons between unordered queries, our approach first sorts the outputs of these queries, and then compares them.

E. Implementation Details

We have implemented our data-oriented testing approach as a Scala command-line tool called CYNTHIA.[1] The interface of CYNTHIA takes as input the names of the ORMs to test along with a set of DBMS on which CYNTHIA runs the ORM code. The tool implements the steps described in Figure 4. For efficiency, CYNTHIA processes testing sessions and ORM queries in parallel using Scala futures [24]. Optionally, CYNTHIA may also receive a random seed (i.e., a number) from the user to make the testing procedure deterministic.

CYNTHIA also provides a *replay* mode, which is used to replay a testing session (i.e., repeat the execution of existing AQL queries) for either debugging purposes or experimenting with different settings (e.g., running existing queries on different DBMSs). Finally, to generate database records, our tool uses the Z3 theorem prover [25], configured with a user-specified timeout.

Regarding the implementation effort of ORM translators, each translator consists of roughly 300–400 lines of Scala code. Every translator traverses the AST of AQL queries and emits code that uses the API of the corresponding ORM. Adding a new translator is guided by extending and implementing an abstract Scala class.

[1] In Greek mythology, Cynthia was the epithet of Artemis, the goddess of the hunt.

TABLE I: The ORM systems examined in our evaluation.

ORM	Language	LoC(k)	Stars(k)	Used By(k)
ActiveRecord (Rails)	Ruby	49.2	46.2	1400
Django	Python	37.7	51.3	466
Sequelize	JavaScript	25.3	22.6	211
SQLAlchemy	Python	150	2.6	182
peewee	Python	7.6	7.7	10

IV. Evaluation

We seek answers to the following research questions:

RQ1 Is CYNTHIA effective in finding new bugs in established ORM systems? (Section IV-B)

RQ2 What are the characteristics of the bugs discovered by CYNTHIA? (Section IV-C)

RQ3 Is solver-based approach effective in generating appropriate data for differential testing? (Section IV-D)

A. Experimental Setup

Target ORM systems. We applied CYNTHIA to the five ORM systems listed in Table I. We selected these ORMs based on the following criteria:

- *Usage*: the ORM should be established and widely-used.
- *High-level Logic*: the ORM should expose a high-level API that abstracts SQL-specific details.
- *Automation*: the ORM must provide tools for easy setup and utilities for generating model classes (recall Section III-A).

According to the Github's statistics, all ORMs incorporated in our evaluation are used by millions of applications. For example, ActiveRecord, which is part of the Rails web framework, is employed by more than 1400k Github repositories. Further, many popular applications and services rely on them. For example, "Nova", OpenStack's cloud computing service, uses SQLAlchemy for interacting with the database. Finally, exposing high-level APIs from programmers can be prone to bugs / errors [26]. Notably, Django, which provides the most expressive API has the most bugs as we will see later.

DBMS. We ran the ORM queries on four DBMSs: SQLite, MySQL, PostgreSQL, and Microsoft's SQL Server (MSSQL). The first three DBMSs are extensively used by the open-source community and are supported by all the examined ORMs. Although MSSQL is supported by a subset of ORMs (i.e., Django, SQLAlchemy, Sequelize), we selected it because is one of the most popular proprietary DBMSs.

Cynthia Configuration. We ran CYNTHIA on a regular basis, and tested the "master" version of the selected ORMs. In each run, CYNTHIA generated five random schemas. After setting up the databases, CYNTHIA spawned a testing session, and processed each testing session separately until a specific timeout was reached (eight hours). For every query, Z3 produced 5 records, while we set the solver timeout to 5 seconds. After each run, we manually inspected the reported mismatches for new bugs, and report them to the developers.

B. RQ1: New Bugs Found

CYNTHIA found 28 bugs in total, out of which, 20 were fixed by the developers, 5 were confirmed but are not yet fixed, 3 are still unconfirmed, while one confirmed bug in Django was previously known and marked as duplicate. Table II summarizes the bug detection results. Django is the system where we detected the most bugs (10), followed by SQLAlchemy (8), Sequelize (5), peewee (4), and finally ActiveRecord (1).

TABLE II: Bugs detected by CYNTHIA.

ORM	Total	Fixed	Confirmed	Unconfirmed
Django	10	6	3	1
SQLAlchemy	8	8	0	0
Sequelize	5	2	1	2
peewee	4	4	0	0
ActiveRecord	1	0	1	0
Total	**28**	**20**	**5**	**3**

TABLE III: The types of the detected bugs and the DBMSs where the bugs manifest themselves.

Type	#Bugs	All DBMS	SQLite	MySQL	PostgreSQL	MSSQL
Logic Error	12	11	0	0	0	1
Invalid SQL	11	3	1	3	2	3
Crash	5	3	0	0	2	0
Total	**28**	**17**	**1**	**3**	**4**	**4**

71% (20 / 28) of the reported bugs have already been fixed by the developers demonstrating the correctness and importance of the reported issues. We were particularly impressed by the prompt fixes of SQLAlchemy and peewee developers: they fixed most of the bugs within six hours after our bug report. Furthermore, three Django bugs were marked as release blockers by the corresponding developers.

C. RQ2: Characteristics of Discovered Bugs

We classify the detected bugs into three categories. The first category (*logic errors*) contains cases where an ORM produced a grammatically and semantically valid SQL query, but this query did not fetch the right data from the database. The second category (*invalid SQL*) contains cases where an ORM yielded either a grammatically or semantically invalid SQL query. The third category (*crashes*) contains cases where an ORM crashed unexpectedly, without even producing an SQL query. Most of the discovered bugs (12) were logic ones (Table III). Unlike differential testing, a naive fuzzing technique is unable to identify such bugs. In a significant number of cases (11), the ORM generated an invalid SQL query, while the remaining cases (5) are related to crashes.

Table III also presents how many bugs are DBMS-dependent. Almost all logic errors (11 / 12) are DBMS-independent, i.e., they appear regardless of the underlying DBMS. By constrast, the majority of "Invalid SQL" bugs are DBMS-dependent. For example, two instances of "Invalid SQL" bugs happen when the code operates on PostgreSQL. Overall, 17 / 28 of the reported bugs are DBMS-independent. Yet, there is a large number of DBMS-dependent bugs (11 / 28). This validates our intuition to test ORMs across multiple database engines.

Based on the feature that ORMs fail to handle correctly, we further classify the discovered bugs into six categories.

Expression-related bugs. Expression-related bugs are the most common ones (7/28). This category involves cases where ORMs fail to produce an SQL expression that respects the

```
1  Comment.new(:rating => 4)
2  Comment.new(:rating => 4)
3  # It incorrectly applies AVG
       to duplicate records.
4  Comment.select("comments.
       rating").distinct.average
       ("comments.rating")
```

(a) A bug in ActiveRecord associated with DISTINCT.

```
1  // WHERE Comment.text LIKE
       '%_%'
2  Comment.findAll({
3   where: {
4    text: {[Op.substring]: "_"}
5   })
6  })
```

(b) A buggy Sequelize query associated with incorrect string comparison.

```
1  cons = ExpressionWrapper(
2    Value(3),...)
3  # GROUP BY Comment.text, 3
4  Comment.objects\
5    .annotate(cons=cons)\
6    .values("cons", "text")\
7    .annotate(sum=Sum("rating"))
```

(c) A buggy Django query associated with GROUP BY.

Fig. 9: A collection of bugs discovered by CYNTHIA.

original ORM query. As an example of this category, consider the peewee bug (Figure 3) discussed in Section III-D. In this bug, peewee produces an SQL expression (i.e., $1+col*1+col$) that is *not* equivalent with the high-level peewee expression written by the programmer (see Figure 3, lines 1, 2).

Distinct-related bugs. DISTINCT is a keyword in SQL that when present, it removes all duplicate records from the result set. ORM systems expose this functionality through a simple method call (typically called `distinct()`). Although the use of this feature looks simple, we detected six bugs related to this functionality. Figure 9a shows a buggy query in ActiveRecord associated with DISTINCT. The intended functionality of this query is to fetch all the records of the table "Comments", remove the duplicates, and then apply AVG to a column named "rating". However, ActiveRecord produces an SQL query that ignores the call of `distinct`, and therefore, it applies AVG to the entire set of records.

Combined-query-related bugs. SQL supports the combination of individual queries using the UNION and INTERSECT keywords. ORM systems support this feature by implementing the `union` and `intersect` methods. Five bugs discovered by CYNTHIA are associated with this functionality of ORMs. An example of this category of bugs has been already discussed in Section III-D (Recall Figure 2). In particular, Django is unable to produce a valid sequence of UNION operations when using MySQL as the database engine.

String-comparison-related bugs. String comparisons in SQL are typically done via the LIKE operator. These operators expect a pattern which SQL matches the value of a string against. There are two characters (namely '%' and '_') that have special semantics when used as part of a LIKE pattern. For example, '%' is a wildcard character that matches any sequence of characters. ORMs typically abstract LIKE with high-level methods, such as `contains()`. ORMs must escape the aforementioned characters when passed as an argument to these methods. We found four cases where ORMs fail to escape these characters leading to wrong string comparisons in the SQL part.

Consider Figure 9b that presents a bug in Sequelize. The Sequelize query shown in this figure attempts to fetch the records of "Comments" where the column "text" contains the character `"_"`. Sequelize produces the SQL condition shown on line 1. Although the character `"_"` has a special meaning (it matches every single character), Sequelize does not escape it. As a result, the generated SQL query incorrectly retrieves all the records of the table.

Aliasing-related bugs. SQL allows column aliasing through the AS construct. ORM systems also support aliasing. CYNTHIA uncovered four bugs where the corresponding ORMs either do not construct the alias correctly, or do not make a reference to a legal alias.

As an example of an aliasing-related bug, consider the SQLAlchemy query: `session.query(Model.column.label("exists"))`. When running this query on SQLite, SQLAlchemy generates the following SQL code: `SELECT "model"."column" AS exists FROM model`. Unfortunately, this SQL query is invalid because "`exists`" is a reserved keyword in SQLite. As a result, the execution of this query throws an *"sqlite3.OperationalError: near "exists": syntax error"* message. To fix this bug, the developers of SQLAlchemy wrapped the reserved word with quotes (i.e., `AS "exists"`).

Group-by-related bugs. The GROUP BY clause is used when selecting or referencing a table's column together with aggregated data. GROUP BY comes with some caveats that ORMs need to consider in order to properly handle this SQL feature. We ran into three bugs caused by incorrect handling of the GROUP BY functionality.

Consider the Django query shown in Figure 9c. Django builds three expressions: the constant 3 (lines 1, 5), a reference to the column "text", and an aggregate function SUM applied to the column "rating". Django places all non-aggregate expressions on GROUP BY as shown on line 3. Integer constants have special semantics when they are part of GROUP BY. For example, GROUP BY 3 means to group by the third expression of the SELECT clause of the query (i.e., `SUM("rating")`). This makes the generated SQL query invalid, leading to *"ProgrammingError: aggregate functions are not allowed in GROUP BY"*. The developers fixed this by ignoring constant expressions from the set of grouping fields.

D. RQ3: Effectiveness of Solver-Based Data Generation

For effectively identifying mismatches between the outputs of ORMs, it is important that ORMs return non-empty results for the given queries. Empty results indicate that the corresponding query was *unsatisfied* with respect to the data inserted to the database. Empty results can potentially hide logic errors that otherwise would be uncovered if the corresponding ORMs could get some data from the database and we were able to notice differences in their results.

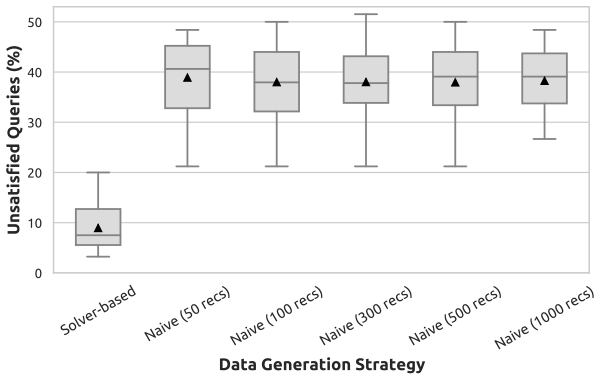

Fig. 10: Percentage of the unsatisfied queries per data generation strategy using a sample of 20 testing sessions.

To demonstrate the effectiveness of our solver-based data generation approach and its suitability for differential testing, we compare it against a simplistic approach that populates the database with random records *a-priori* [21], [27], i.e., it inserts data while setting up the tables, without considering the constraints of the generated queries.

We used CYNTHIA to spawn 20 testing sessions. For each testing session, we generated 100 queries and compared the results of ORMs as usual. At the end of each testing session, we measured in how many queries the ORMs returned empty results. We then *replayed* each testing session, using a naive data generation strategy, and tried out different settings: generating 50 random records, 100, 300, 500 and finally 1000.

Figure 10 illustrates the comparison results. The y-axis shows the percentage of the unsatisfied queries. Every box plot contains the observations taken from the 20 testing sessions, along with the median (horizontal line), the mean (black triangle), and the maximum and minimum values. The solver-based approach leads to significantly fewer unsatisfied queries (median: 7.5%, mean: 8.9%) than the naive approach (mean and median values are roughly 38% for all the different settings). The reason why there is still a number of unsatisfied queries even with the solver-based approach is because either the corresponding AQL query was unsatisfiable or the solver timed out. Regarding the naive data generation, increasing the number of the records inserted to the database does not improve the effectiveness of this method at all, i.e., generating 50 records is almost identical to generating 1000 records.

We also tried to reproduce the discovered bugs using the naive data generation strategy. This strategy missed 3 out of the 12 logic errors previously detected by CYNTHIA, because it failed to generate appropriate data for the database. In these cases, the differential testing was meaningless, as the ORMs returned empty results. We did not consider the rest categories (e.g., invalid SQL), as in these cases the corresponding ORMs produce an error message regardless of the data stored in the database. Overall, our findings suggest that it is the quality of the inserted data that matters, and not the quantity: it is better to produce 5 targeted records than 1000 random records.

E. Discussion & Threats to Validity

Regression Bugs. Running CYNTHIA on the master version of ORMs enabled us to find a couple of interesting regression bugs. Regression bugs indicate that a feature that worked properly in previous versions, is broken in the current implementation. These bugs were of paramount importance for the developers. For example, Django developers marked our regression bugs as release blockers. Also, SQLAlchemy developers commented: *"it's very useful if you are in fact alpha testing it."* (i.e., master branch). We also noticed that some bugs that were allegedly fixed were triggered again by new queries. This observation was confirmed by the developers, who, indeed reopened and fixed old bugs reported by us.

ORMs. Although it is the de-facto framework for Python, the Django ORM is the system where our approach detected the most bugs. One may wonder why we detected so many bugs in Django, while we uncovered only one bug in ActiveRecord. The reason is that Django is a more high-level ORM than ActiveRecord: it hides *every* single SQL-detail via its API. On the other hand, ActiveRecord's API provides some functionalities that are closer to SQL. For example, ActiveRecord supports arithmetic operations and aliasing by writing plain SQL. Thus, ActiveRecord does not employ any sophisticated translation mechanism and in many cases, the input of the programmer is passed directly to the SQL code.

DBMSs. CYNTHIA identified four PostgreSQL- and MSSQL-related bugs. These ORM bugs are triggered only when the DBMS is switched to PostgreSQL or MSSQL. On the other hand, only one ORM bug is related to SQLite. This happens because PostgreSQL and MSSQL are much stricter than SQLite (and even MySQL). For example, unlike MySQL and SQLite, PostgreSQL has a strict type system, and comes with many restrictions that ORMs need to take into account when producing SQL code. Also, we note that during our testing efforts, we discovered one bug in SQLite. The bug was already known and fixed in a later version of SQLite though. This implies that with some tuning, our approach may be also useful for testing DBMSs.

Threats to Validity. A threat to the internal validity of our approach, involves correctness bugs in the implementation of our translators. In this case, a mismatch in ORM results may be caused due to a bug in our translators and not in ORMs themselves. To mitigate this threat, before reporting a bug to the developers, the first two authors carefully examined each mismatch to verify that it was not generated by an error in the translators.

A threat to external validity is related to the representativeness of the examined ORMs. All the selected ORMs are popular and used by millions of applications. There is no fundamental limitation on supporting other ORM implementations. As our prototype gains developer traction, we will implement more translators (e.g., for JPA implementations).

Finally, a threat to construct validity concerns the generalizability of our approach. Our approach targets to find bugs associated with the translation of ORM API calls into SQL queries. An ORM though, may suffer from other kinds of bugs, such as performance issues, transaction management, or configuration of model classes.

V. Related Work

Quality in ORM-based Applications. A number of tools and studies have been proposed to improve the quality of ORM-based applications. Chen et al. [2] introduced a static analysis framework for identifying ORM queries in Java applications that degrade the response times of database engines. Their approach first explores the paths of the program to identify database accesses, and then detects performance anti-patterns through a rule-based approach. Furthermore, their technique provides an assessment mechanism for prioritizing the fixes of the detected performance issues. Subsequent work [28], [29] focused on fixing performance issues through automated means. In particular, Singh et al. [29] introduced a genetic algorithm for tuning the configuration of ORM systems to achieve better performance. Davar et al. [28] proposed a refactoring framework by applying a set of known transformation rules to inefficient ORM-based code. Unlike prior work that finds issues in the ORM-based applications, our work is the first to find issues in the ORM implementations.

Testing of DBMSs. The work of Slutz [30] is the first to uncover bugs in DBMSs using a differential testing approach. To safely compare results, his method generates random queries on a small subset of the SQL language that is common across DBMSs. Over the past decade, there have been numerous approaches for generating (targeted) SQL queries in order to effectively test DBMSs [31], [32], [33], [34]. The most recent approaches are SQLsmith [35] and SQLfuzz [36], two SQL query generators that respectively target crashes and regression bugs in popular DBMSs. Our approach differs from all these query generators because it produces queries in a higher-level query language (AQL) and adopts differential testing to detect logic errors beyond crashes or regression bugs. Khalek et al. [37], [34] followed a solver-based approach for testing DBMSs. Their work employs a relational constraint solver to generate valid database records with respect to a given SQL query and database schema. Besides populating the database, their method also determines the expected results of an SQL query and the authors use this oracle to find bugs. We also use an SMT-solver to populate the database, but we specify the test oracle by adopting a differential testing approach.

More recently, Rigger et al. [21] proposed the *Pivoted Query Synthesis* (PQS) technique for testing database engines. PQS generates SQL queries so that they fetch a specific record from the database. In this way, PQS forms the test oracle: failing to fetch the expected record reveals a potential bug in DBMS. Unlike this work, our approach adopts differential testing for determining the oracle. Also, beyond reasoning about a single record, our approach is able to detect bugs involving operations on result sets (e.g., aggregate functions, sorting, distinct). In an attempt to find optimization bugs in database systems, their subsequent work introduced a metamorphic testing technique called *Non-Optimizing Reference Engine Construction* (NoREC) [27]. At a high-level, NoREC applies a semantics-preserving transformation to a given SQL query in way that the various optimizations performed by the DBMS are disabled. Finally, NoREC compares the results of the original and the resulting queries for mismatches. In their most recent work, they propose *Ternary Logic Partitioning* (TLP) [38]. Given an SQL query, TLP derives multiple queries that compute a partial result of the initial query, and then combines the results of each individual query using a `UNION` operation. If the result of the combined query does not match that of the initial one, then a bug is found. TLP is suitable for testing the implementation of the `WHERE`, `HAVING`, `DISTINCT` clauses, or aggregate functions.

All these previous approaches are tailored to testing DBMSs, i.e., they aim to find DBMS-specific bugs (e.g., optimization bugs, bugs associated with the evaluation of `WHERE` clauses). ORM systems differ from database engines, and suffer from other types of bugs.

Differential Testing. Differential testing [22], [39] is a generally-applicable testing technique that aims to find bugs in software implementations by addressing the oracle problem [40]. Differential testing has been successfully applied to various domains, most notably compilers and runtime systems [16], [19], [17], [18], [20]. Following this success, differential testing has been applied to many other domains, from program analyzers, such as model checkers [41], debuggers [42], and symbolic execution engines [43], to probabilistic programming languages [44], and software libraries and services [45], [46], [47], [48]. Inspired by this work, we also employ differential testing for finding bugs in ORM systems.

VI. Conclusion

A fundamental requirement for differential testing is that the implementations under test must be equivalent. By introducing an appropriate layer of abstraction that hides the implementation differences (AQL), we showed that differential testing can be also applicable in systems with (seemingly) dissimilar interfaces, such as ORMs.

Further, we addressed an ORM-specific challenge: the generation of data that are likely to produce non-trivial results in response to given queries. To do so, we employed an SMT solver to synthesize targeted records, dependant on the constraints of the generated inputs. Our findings showed that when compared to other simplistic data generation strategies, the solver-based approach enhances the bug detection capability.

We demonstrated the importance and practicality of our approach by systematically testing five popular open-source ORM systems. We discovered 28 bugs, most of which have been fixed by the developers. The effectiveness of our method can be further improved by considering other forms of queries and functionalities, such as insert or update operations, and transaction management.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. We also thank the developers of the examined ORM systems for addressing our bug reports. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 825328.

REFERENCES

[1] A. Torres, R. Galante, M. S. Pimenta, and A. J. B. Martins, "Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design," *Information and Software Technology*, vol. 82, pp. 1 – 18, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584916301859

[2] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 1001–1012. [Online]. Available: https://doi.org/10.1145/2568225.2568259

[3] K. Roebuck, *Object-Relational Mapping (ORM): High-Impact Strategies-What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors*. Emereo Publishing, 2012.

[4] C. Bauer, G. King, and G. Gregory, *Java Persistence with Hibernate*, 2nd ed. USA: Manning Publications Co., 2015.

[5] D. Maier, *Representing Database Programs as Objects*. New York, NY, USA: Association for Computing Machinery, 1990, p. 377–386. [Online]. Available: https://doi.org/10.1145/101620.101642

[6] M. Eltsufin, "Bringing Hibernate ORM to cloud Spanner for database adoption," https://cloud.google.com/blog/products/databases/bringing-hibernate-orm-cloud-spanner-database-adoption, 2019.

[7] T. Chen, W. Shang, J. Yang, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, "An empirical study on the practice of maintaining Object-Relational Mapping code in Java systems," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 165–176.

[8] M. Bayer, "SQLAlchemy - the database toolkit for Python," https://www.sqlalchemy.org/, 2020, [Online; accessed 29-July-2020].

[9] D. S. Foundation, "The web framework for perfectionists with deadlines," https://www.djangoproject.com/, 2020, [Online; accessed 29-July-2020].

[10] J. Yang, P. Subramaniam, S. Lu, C. Yan, and A. Cheung, "How ¡i¿not¡/i¿ to structure your database-backed web applications: A study of performance bugs in the wild," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 800–810. [Online]. Available: https://doi.org/10.1145/3180155.3180194

[11] D. S. Foundation, "Django issues," https://code.djangoproject.com/query, 2020, [Online; accessed 29-July-2020].

[12] A. Viswa, "select_for_update() with "of" uses wrong tables from (multi-level) model inheritance," https://code.djangoproject.com/ticket/31246, 2020, [Online; accessed 29-July-2020].

[13] S. Bank, "negated EXISTS result type not bool with SQLite dialect," https://github.com/sqlalchemy/sqlalchemy/issues/3682, 2016, [Online; accessed 29-July-2020].

[14] "CVE-2019-7164," https://nvd.nist.gov/vuln/detail/CVE-2019-7164, 2019, [Online; accessed 29-July-2020].

[15] "CVE-2020-9402," https://nvd.nist.gov/vuln/detail/CVE-2020-9402, 2020, [Online; accessed 29-July-2020].

[16] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[17] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of JVM implementations," *SIGPLAN Not.*, vol. 51, no. 6, p. 85–99, Jun. 2016. [Online]. Available: https://doi.org/10.1145/2980983.2908095

[18] Y. Chen, T. Su, and Z. Su, "Deep differential testing of JVM implementations," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 1257–1268. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00127

[19] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," *SIGPLAN Not.*, vol. 50, no. 6, p. 65–76, Jun. 2015. [Online]. Available: https://doi.org/10.1145/2813885.2737986

[20] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 203–213. [Online]. Available: https://doi.org/10.1145/2884781.2884879

[21] M. Rigger and Z. Su, "Testing database engines via pivoted query synthesis," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 667–682. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/rigger

[22] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[23] D. S. Foundation, "Integrating Django with a legacy database," https://docs.djangoproject.com/en/3.0/howto/legacy-databases/, 2020, [Online; accessed 29-July-2020].

[24] P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic, "Futures and promises," https://docs.scala-lang.org/overviews/core/futures.html, 2020.

[25] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[26] V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh, "POSIX abstractions in modern operating systems: The old, the new, and the missing," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: Association for Computing Machinery, 2016.

[27] M. Rigger and Z. Su, "Detecting optimization bugs in database engines via non-optimizing reference engine construction," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1140–1152. [Online]. Available: https://doi.org/10.1145/3368089.3409710

[28] Z. Davar and Handoko, "Refactoring object-relational database applications by applying transformation rules to develop better performance," in *Proceedings of the 16th International Conference on Information Integration and Web-Based Applications & Services*, ser. iiWAS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 283–288. [Online]. Available: https://doi.org/10.1145/2684200.2684304

[29] R. Singh, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Optimizing the performance-related configurations of object-relational mapping frameworks using a multi-objective genetic algorithm," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 309–320. [Online]. Available: https://doi.org/10.1145/2851553.2851576

[30] D. R. Slutz, "Massive stochastic testing of SQL," in *Proceedings of the 24rd International Conference on Very Large Data Bases*, ser. VLDB '98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, p. 618–622.

[31] N. Bruno, S. Chaudhuri, and D. Thomas, "Generating queries with cardinality constraints for DBMS testing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 12, pp. 1721–1725, 2006.

[32] H. Bati, L. Giakoumakis, S. Herbert, and A. Surna, "A genetic approach for random testing of database systems," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007, p. 1243–1251.

[33] C. Mishra, N. Koudas, and C. Zuzarte, "Generating targeted queries for database testing," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 499–510. [Online]. Available: https://doi.org/10.1145/1376616.1376668

[34] S. Abdul Khalek and S. Khurshid, "Automated SQL query generation for systematic testing of database engines," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 329–332. [Online]. Available: https://doi.org/10.1145/1858996.1859063

[35] A. Seltenreich, "SQLsmith: A random SQL query generator," https://github.com/anse1/sqlsmith, 2020, [Online; accessed 29-July-2020].

[36] J. Jung, H. Hu, J. Arulraj, T. Kim, and W. Kang, "APOLLO: Automatic detection and diagnosis of performance regressions in database systems," *Proc. VLDB Endow.*, vol. 13, no. 1, p. 57–70, Sep. 2019. [Online]. Available: https://doi.org/10.14778/3357377.3357382

[37] S. Abdul Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid, "Query-aware test generation using a relational constraint solver," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 238–247.

[38] M. Rigger and Z. Su, "Finding bugs in database systems via query partitioning," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428279

[39] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "NEZHA: Efficient domain-independent differential testing," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 615–632.

[40] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[41] C. Klinger, M. Christakis, and V. Wüstholz, "Differentially testing soundness and precision of program analyzers," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 239–250. [Online]. Available: https://doi.org/10.1145/3293882.3330553

[42] D. Lehmann and M. Pradel, "Feedback-directed differential testing of interactive debuggers," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 610–620. [Online]. Available: https://doi.org/10.1145/3236024.3236037

[43] T. Kapus and C. Cadar, "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*, ser. ASE 2017. IEEE Press, 2017, p. 590–600.

[44] S. Dutta, O. Legunsen, Z. Huang, and S. Misailovic, "Testing probabilistic programming systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 574–586. [Online]. Available: https://doi.org/10.1145/3236024.3236057

[45] S. Srisakaokul, Z. Wu, A. Astorga, O. Alebiosu, and T. Xie, "Multiple-implementation testing of supervised learning software," in *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[46] M. Selakovic, M. Pradel, R. Karim, and F. Tip, "Test generation for higher-order functions in dynamic languages," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276531

[47] Y. Chen and Z. Su, "Guided differential testing of certificate validation in SSL/TLS implementations," ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 793–804. [Online]. Available: https://doi.org/10.1145/2786805.2786835

[48] P. Godefroid, D. Lehmann, and M. Polishchuk, "Differential regression testing for REST APIs," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 312–323. [Online]. Available: https://doi.org/10.1145/3395363.3397374