Generating Test Databases for Database-Backed Applications

Cong Yan
Microsoft Research
Redmond, USA
cong.yan@microsoft.com

Suman Nath
Microsoft Research
Redmond, USA
suman.nath@microsoft.com

Shan Lu
University of Chicago
Chicago, USA
shanlu@uchicago.edu

***Abstract*—Database-backed applications are widely used. To effectively test these applications, one needs to design not only user inputs but also database states, which imposes unique challenges. First, valid database states have to satisfy complicated constraints determined by application semantics, and hence are difficult to synthesize. Second, the state space of a database is huge, as an application can contain tens to hundreds of tables with up to tens of fields per table. Making things worse, each test involving database operations takes significant time to run. Consequently, unhelpful database states and running tests on them can severely waste testing resources.**

We propose DBGRILLER, a tool that generates database states to facilitate thorough testing of database-backed applications. To effectively generate valid database states, DBGRILLER strategically injects minor mutation into existing database states and transforms part of the application-under-test into a stand-alone validity checker. To tackle the huge database state space and save testing time, DBGRILLER uses program analysis to identify a novel *branch-projected DB view* that can be used to filter out database states that are unlikely to increase the testing branch coverage. Our evaluation on 9 popular open-source database applications shows that DBGRILLER can effectively increase branch coverage of existing tests and expose previously unknown bugs.

***Index Terms*—Automated testing, Test data generation, database-backed application, database-state generation**

I. INTRODUCTION

Database-backed applications, which store application states in persistent databases, are ubiquitous. Many of today's most important and most popular applications are database-backed.

Thorough testing of a database-backed application is challenging as its behavior depends not only on traditional user inputs, but also on the underlying database states. Our study of 30 popular database-backed applications (details in §III) shows that the conditions of more than half of the branches in these applications are affected by underlying databases. To ensure that existing tests cover these *DB-dependent branches*, developers manually create various test databases. However, given the large number of DB-dependent branches, developers often struggle to create a sufficient number of test databases that can ensure a good coverage of the DB-dependent branches (and thus find bugs in those branches). In our aforementioned study, existing tests, with developer provided test databases, cover only 55% of the DB-dependent branches.

This paper aims to automatically synthesize test databases that can increase DB-dependent branch coverage of existing tests. This problem is challenging for two key reasons.

First, *validity*. Synthesized databases need to be valid according to the application semantics. The underlying database of an application is typically populated by the application itself, and hence it can contain only semantically meaningful states. For example, in an enterprise application, an employee's salary cannot be negative and an employee name in the Salary table must also appear in the Employee table. With a semantically invalid database, a test may fail due to invalid data, without revealing application bugs (i.e., a false positive).

Second, *efficiency*. Popular database-backed applications typically contain tens to hundreds of tables, with up to tens of fields per table. Therefore, the number of possible test databases is huge, if not unlimited. Making things worse, running a test on each database state takes non-negligible amount of time due to expensive database operations (more than 10 seconds per test in our experiments).

These challenges particularly affect the effectiveness of traditional fuzzing techniques [1], [39] that would waste much computation resource in testing many database states that are invalid or unhelpful in improving test coverage. In theory, one may use symbolic execution and constraint solving to synthesize database states that are feasible to be produced by the application and also capable of improving test coverage. However, given the complexity of popular database-backed applications and their extensive use of third-party libraries [41], this approach would be intractably expensive.

We address these challenges with a novel solution that leverages existing features of database-backed applications. To address the *validity* challenge, we use two mechanisms. First, we use database states that are designed by developers for existing tests as *seed states* and *mutate* them to generate new test databases. For example, we replace a non-null value in a seed state with a null value, which may increase the coverage of branches conditioning on whether the value is null or not. We propose five mutation mechanisms to synthesize new databases that, as our experimental results show, are mostly valid. This strategy relieves us from the daunting task of synthesizing valid databases from scratch. Second, to discard the small number of invalid databases, we leverage the fact that database-backed applications commonly contain functions to

validate semantic correctness of their database updates. We use static analysis to extract those validation functions and combine them into a stand-alone checker function that can efficiently identify invalid databases.

In our experiments, the above process easily generates tens of thousands to hundreds of thousands of valid database states per application, and running existing tests on all of them can take several months. We therefore face the *efficiency* challenge.

To address the *efficiency* challenge, we use program analysis to filter out database states that are unlikely to improve the testing branch coverage. The insight is that only a subset of database tables and fields can potentially improve branch coverage, by affecting the conditional predicates of uncovered branches. We can use program analysis to identify the subset. Given a database state, we refer to the values of this subset of tables/fields as *branch-projected view* of the database. For the purpose of code coverage, two database states with the same branch-projected view are equivalent (since the values outside the view do not affect branch conditions). This enables us to identify and filter out redundant database states that have the same branch-projected view as an already considered database state (and hence they are unlikely to improve the testing branch coverage). The technique is effective (it discards 90% of the synthesized databases in our experiments) and efficient (an order of magnitude faster than executing the checker or a test), making the entire pipeline of database generation + validation + test highly efficient.

We have implemented the techniques in a system called DBGRILLER, and evaluated DBGRILLER on nine popular open-source Ruby applications. Our evaluation shows that, compared with the database states originally designed by application developers, DBGRILLER covers 25% more DB-dependent branches on average (up to 63%), increasing the overall coverage of DB-dependent branches from 42–69% to 51–80% across these nine applications. Among all DB-dependent if-else branch pairs that are partially covered with original database states (i.e., only the `if` branch or only the `else` branch is covered), 35% (up to 52%) become fully covered after using DBGRILLER. Although DBGRILLER is not a bug finding tool by itself, its increased branch coverage may expose more bugs with suitable test oracles. In evaluation, using a simple test oracle catching 404 webpage, DBGRILLER identifies 22 unique bugs that are not exposed by existing tests.

II. BACKGROUND

Database-backed applications commonly use the Model-View-Controller (MVC) architecture, where each user request triggers a *controller* action. For instance, a checkout request through the URL `https://foo.com/checkout?uid=1&oid=2` triggers a controller with the request parameters `uid` and `oid`. Inside a controller action, the application interacts with back-end database via an Object-Relational Mapping (ORM) library such as `ActiveRecord` in Rails. The ORM library translates database-related tasks into SQL queries and issues them to the database. The library also serializes query results into a *model* object (e.g., an `ActiveRecord` object), which is then used by the application to generate a response. Thus, the ORM library enables applications to work on model abstractions, instead of directly interacting with underlying databases.

```
1 class Order < ActiveRecord
2 # constraint checker for Order
3   validate: validate_email
4   def validate_email
5     if email.nil? || email.blank?
6       return error("email address missing in order")
7     end
8   end
9 end
10 # interaction with database through ORM library
11 order = Order.find_by_id(param[:order_id])
12 order.line_items.each do |item|
13   # DB-dependent branches
14   if Inventory.find_by_id(item.inventory_id).count() >=
         item.quantity
15     render :successful_checkout(order)
16   else
17     render :insufficient_inventory(order,item)
18 end
```

Listing 1. An example test code snippet, abridged from Spree[18]

Often, data stored in the database needs to satisfy certain constraints. Such constraints are expressed either in the database or in the application. This is illustrated with an example shown in Listing 1, which is abridged from Spree[18]. Here the constraint checker ensures that the `name` field of the `Order` table is not null or empty (Line 3-8). The listing also shows how the application interacts with the database through an ORM function (`find_by_id` in Line 11) and how the data in the database affects branch coverage.

```
1 # populate test database
2 before do
3   c = create(User, :id=1, :pass='1234', ...)
4   o = create(Order, :id=2, :token='abc', ...)
5   i = create(LineItem, :id=1, :order_id=2, ...)
6 end
7 # run test with the test database
8 response=post("https://foo.com/checkout?uid=1&orderid=2")
9 expect(response.status).to eq(200)
```

Listing 2. An example application test

Developers commonly write end-to-end tests that take URLs with parameters as inputs and check assertions on response. Because a test needs to interact with an underlying database, a developer populates the database before the test starts with carefully designed values that we call *DB-state*. An example is shown in Listing 2 where the developer first populates the database with three tables (Lines 2-7) and then issues a post query to be executed with the test database (Line 9).

Note that a DB-state can affect the code coverage of a test. For example, in Listing 1, the database value read in Line 11, 12 and 14 affects how the following *DB-dependent branches* execute. As we will show later, developer-provided DB-states often achieve poor coverage of DB-dependent branches, a problem that we aim to address with DBGRILLER.

III. EXTENDED MOTIVATION

In this section, we motivate our problem and solution by analyzing popular open-source database-backed apps built with Ruby on Rails, a popular framework to build such apps. We use a combination of static and dynamic analysis techniques. We statically analyze (details in § VI-B1) 30

TABLE I
DETAILS OF THE APPLICATIONS

Name	Abbr.	Category	Stars	LoC	Database	# Table	# Field	# Constraints	# test	# branch*	# DB-branch
Forem[7]	Fr	Forum	18.8k	158K	Postgres	84	922	890	1852	5478	3126 (57%)
Lobsters[11]	Lb	Forum	3.2k	20K	MySQL	23	165	243	67	1024	564 (55%)
Chatwoot[3]	Ch	Customer service	12.1k	35K	Postgres	48	374	412	374	1068	554 (52%)
Spree[18]	Sp	Online shopping	11.7k	52K	Sqlite	83	648	645	438	3870	2044 (53%)
Tracks[20]	Tr	Task mgmt	1.1k	29K	MySQL	17	136	73	232	2416	1206 (50%)
Huginn[10]	Hg	Event tracking	35k	48K	MySQL	11	113	76	155	1828	998 (55%)
Openstreetmap[12]	Om	Map service	1.4k	104K	Postgres	57	355	479	462	3082	1286 (41%)
AutoLab[2]	Al	Homework grading	600	30K	MySQL	26	241	109	103	2262	1388 (61%)
GrowStuff[8]	Gs	Farming mgmt	365	33K	Postgres	43	334	221	144	1204	696 (58%)

*We follow the standard branch counting and an `if-else` block contains two branches.

popular applications with > 350 stars on GitHub. Our dynamic analysis (and evaluation of DBGRILLER) is based on a smaller subset of 9 applications (shown in Table I) that could be set up locally to run tests with reasonable efforts,[1] and have been actively maintained in last one year. These 9 applications are diverse in category (including forum, online shopping, task management, etc.), highly starred, developed for years, and deployed to serve a large number of users everyday.

Q1: How important is DB-state for testing applications? Our static analysis shows that over 98% of end-to-end tests interact with database state (by using ORM APIs such as in Line 11 of Listing 1). Moreover, 52% of all branches are DB-dependent; i.e., their conditions depend on DB-states. These numbers show that DB-states are crucial for tests and increasing their coverage in database-backed applications. However, our dynamic analysis shows that existing tests consider specific scenarios (and associated DB-states) and cover mostly below 60% of the DB-dependent branches (details in § VII-B). This motivates for a tool to improve such coverage.

Q2: How complex is the application code? Rail's ActiveRecord provides over 70 methods to interact with the database [15], meanwhile we observe many customized SQL queries used in these 9 applications. Furthermore, we observe that each application uses over 30 (up to 157) libraries whose source code may be unavailable for analysis. These complexities, adding to the dynamically-typed nature of Ruby, *makes it difficult to apply either static analysis or symbolic-execution and constraint solving to generate DB-states*.

Q3: How long does it take to run a test and how complex is a database state? End-to-end tests on database are slow: in the 9 apps listed in Table I, a test takes from 5 to 25 seconds to finish, with the majority taking over 12 seconds. *This prohibits the use of traditional fuzzing techniques* that assume thousands of invocations per second and hence can try many tests quickly. The space of possible database states is also huge, if not infinite. We find that each application has 11 to 84 tables, with up to a total of 922 fields in one app. Moreover, each includes from 73 to 890 data constraints specified in the application code and the database. While some constraints are generic (e.g., unique, not null, and foreign-key constraint), some are complicated and written in Ruby code (e.g., returned item does not overlap with final sale items), making them hard to model. Due to the large space of database states and complexity of data constraints, *it is extremely challenging to design application-specific DB-state fuzzing technique* that can efficiently explore the space and generate valid DB-states.

[1]For example, we could not dynamically analyze Diaspora[5]. It requires setting up multiple machines with public facing URLs which is non-trivial to configure, and the app fails to run tests even after following its installation instructions.

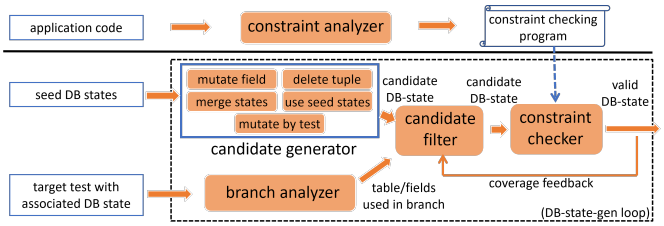

Fig. 1. Workflow of DBGRILLER.

Q4: How prevalent are tests and constraints? 27 out of the 30 apps we studied include a good number of end-to-end tests: ranging from 67 to 3468 tests and an average of 561 tests per app. Each test includes a DB-state. Applications also include a large number of data constraints, ranging from 73 to 890. This is encouraging since DBGRILLER aims to leverage existing tests, DB-states, and constraints.

IV. WORKFLOW OF DBGRILLER

For a given test $\mathcal{T}$, we have three design goals for DBGRILLER: (1) *validity*: feed to $\mathcal{T}$ not only syntactically but also semantically valid database states (DB-states), as testing results on invalid database states are not trustworthy; (2) *effectiveness*: increase branch coverage of $\mathcal{T}$; and (3) *efficiency*: synthesize few DB-states while satisfying the first two goals to save testing resources.

DBGRILLER has three key innovations to achieve these goals, reflected in its workflow shown in Figure 1. First, to achieve the *validity* goal, DBGRILLER re-purposes data validation functions that already exist in applications, rewriting them into a constraint checkers that can validate any given DB-state. Second, to achieve the *effectiveness* goal, we design a series of mechanisms that leverage DB-states created by developers to generate new DB-states that are likely to be valid and can effectively increase coverage of DB-dependent branches. Third, to achieve the *efficiency* goal, we design a novel algorithm to filter out DB-states that are not useful to

increase branch coverage. The algorithm computes a *branch-projected view* which contains tables and fields used by conditions of uncovered branches in $\mathcal{T}$ and filters out DB-states with duplicate branch-projected views.

V. Constraint Checker

As observed by prior works[46], database-backed applications contain many data-validation constraints that are invoked before data is stored in database. For example, while creating a new account, an application may use a constraint to ensure that the account id is unique before storing the account information in the database. DBGriller extracts such constraints that are scattered around in an application and wraps them into a standalone checker, to be used later to validate a DB-state generated by DBGriller. This requires addressing two key questions: (1) how to identify constraints checking code in an app? and (2) what tables are checked with what constraints?

To answer the first question, we leverage the observation from prior work [46] that most application-level constraints are expressed in *validation callbacks*. This callback is invoked every time a tuple is inserted/updated into the database, and throws an exception if the tuple fails the validation logic specified by developers. An example is shown in Listing 1 at Line 2-9: developers define a validation callback, `validate_email`, that is invoked every time an order record is saved and returns an `error` if the order's email field is `NULL` or blank. DBGriller extracts all such validation callback functions through static analysis using the method described in [46].

We answer the second question with the observation that applications using ORM frameworks often implement operations on a database table inside a Model class whose name matches the table name. For example, the validation code in Listing 1 that checks a constraint on the `orders` table is implemented with the `Order` class. This enables DBGriller to identify the set of validation functions for each table (validation involving multiple tables is implemented in one class model whose invokation automatically extracts data from all other tables).

Once all the validation functions and their relevant tables are identified, DBGriller wraps them in a constraint checker program. Given a DB-state, it iterates over all tables, identifies the set of validation functions defined for that table, and then invokes each validation on all tuples of the table. It returns the set of validator functions failing on the input DB-state. An empty return value indicates a successful validation.

Filtering irrelevant constraints. It is possible that a constraint C is irrelevant to a test $\mathcal{T}$, e.g., when C involves tables or columns that are not accessed by $\mathcal{T}$. In such a case, violation of C should not prevent a DB-state from being used by $\mathcal{T}$. For instance, in application *Chatwoot*, one validation function V checks that every message inbox is associated with an account (i.e., `account_id` of `inboxes` table is not null and the corresponding tuple in table `accounts` exists). However, a test may not query the `account` table and do not use the `account_id` field. To minimize effort, developers design a DB-state including `inboxes` tuples but empty `account` table for these tests, on which the tests still run successfully.

To allow these invalid DB-states whose failed constraints do not affect test $\mathcal{T}$, we run the constraint-checking procedure on the DB-state defined by the developer for $\mathcal{T}$. Any constraints that are violated by this original DB-state are considered as not required for $\mathcal{T}$. These constraints are then ignored while validating DB-states generated for $\mathcal{T}$.

VI. Database State Generator

This section describes our algorithms to produce DB-states for a given test $\mathcal{T}$, with the workflow shown in Figure 1.

A. Candidate Generation

The goal of this step is to generate DB-states that are likely to be (1) valid according to the application constraints, and (2) diverse to increase code coverage. These enable DBGriller to avoid many unnecessary invocations of expensive checkers on invalid DB-states and of $\mathcal{T}$ on DB-states that do not increase branch coverage.

Generating such DB-states from scratch is non-trivial. One might consider leveraging the schema of the database to generate syntactically correct DB-states. However, given the complicated data-semantic constraints and the complicated application-database relationship, such a state is very likely to be semantically invalid or unhelpful in increasing branch coverage. We experimentally confirm this in §VII-D. We therefore leverage the large number of seed DB-states developers have already created for $\mathcal{T}$ and other tests of the applications and mutate them to generate new states. Such new states are likely to be valid since seed states are valid. Moreover, they are likely to be diverse since they are derived from seed-states defined for not only $\mathcal{T}$, but also for other tests of the application. These seed-states provide not only diverse field values but also diverse relationships between table rows (like foreign keys) that are important to the application.

We have designed five mechanisms to generate new DB-states from seed DB-states. The first two mechanisms, mutating table fields and deleting tuples, directly mutate the DB-state the developer has created for $\mathcal{T}$ (denoted as $state_T$). They target increasing coverage of branches that condition on specific field value or empty/non-empty query results. However, they alone only produce few mutations, and hence we use seed DB-states to bring more changes beyond a single field or a single tuple. We have three additional mechanisms for this purpose. We merge states to bring tuples from seed database to $\mathcal{T}$'s DB-state, use seed DB-state directly and obtain mutated seed DB-states by running tests on them. As we will show in §VII-C, these mechanisms produces highly-likely valid states, individually increasing branch coverage yet complimentary to each other, achieving much higher coverage altogether. The five mechanisms are described as follows.

1) Mutating table field: This mechanism starts with the DB-state $state_T$ for $\mathcal{T}$ as defined by the developer and mutates the value of every table field to increase the chance of covering branches whose condition depends on a specific field value, like the `quantity` field Listing 1 Line 14. Similar to existing fuzzing tools, new values are obtained based on predefined

rules for the type of a field. For fields with limited value space like *boolean* and *enum* fields, or fields with an inclusion constraints (meaning that its value can only be one of a set of predefined values), all values in this space will be used to update that field. For other fields like *integer*, *float* or *string*, DBGRILLER chooses 5 values: a `NULL` value, one special value (`""` for string field and 0 for numerical field), one random value sampled from corresponding value space and two values randomly chosen from the same field of other seed states. For each new value, DBGRILLER issues an `UPDATE` query on $state_T$ to obtain a candidate DB-state. Listing 3 shows the 5 mutating-field queries mutating `email` field of string type from the `orders` table, resulting in 5 candidates.

```
UPDATE emails SET name = NULL;
UPDATE emails SET name = ''; //empty string
UPDATE emails SET name = 'xkosk2ldo'; //random value
UPDATE emails SET name = 'foo at gmail.com'; //from seed
UPDATE emails SET name = 'bar at yahoo.com'; //from seed
```

Listing 3. Example queries generated by mutating field

2) Deleting tuple.: DBGRILLER deletes existing tuples in $state_T$ to increase the chance of covering branches that depend on empty/non-empty query result. DBGRILLER issues `DELETE` query with primary key to delete one tuple from $state_T$ each time to generate a candidate DB-state. To ensure a successful deletion, it nullifies all foreign key reference to that tuple or deletes the referencing tuple if the foreign key field has a not-null constraint. For instance, when deleting an `orders` tuple which is referenced by a `line_items` tuples belonging to that order, DBGRILLER issues two queries as shown in Listing 4.

```
UPDATE line_items SET order_id=NULL WHERE order_id=1;
DELETE orders WHERE id = 1;
```

Listing 4. Example queries generated by deleting tuple

3) Merging DB-states.: Tuple deletion may cover branches that depend on empty query results, yet covering branches on non-empty results may require adding new tuples to $\mathcal{T}$. Since generating a new tuple with random values is likely to fail the data constraint, DBGRILLER adds new tuples to $state_T$ from other seed DB-states. To do so, it randomly selects a few seed DB-states (20 in our prototype) to merge into $state_T$, runs an `INSERT` query for each tuple with `IGNORE` keyword to skip insert failures (for instance, failure due to duplicated primary key). In our prototype, DBGRILLER generates 10 candidate states using this method. We also experimented with different numbers of candidates and found that generating more than 10 candidates adds limited values while producing states with duplicated views.

4) Using other seed DB-states.: This mechanism simply uses a seed DB-state that is different from $state_T$. Since seed DB-states are designed by developers, they are likely to be valid. Using a DB-state different from $state_T$ is likely to increase the branch coverage.

5) Mutating seed DB-states by running test.: This mechanism expands the pool of seed DB-states by running tests on the seed DB-state. When a test modifies DB-states, it often brings new interesting tuples or field values that not seen in the seed DB-states originally designed by developers. Because each seed DB-state is designed for one particular test, this mechanism simply runs each test on their associated database to obtain a new candidate DB-state.

B. Candidate filtering

The above mechanisms can potentially generate a prohibitively large number of candidates, yet most of them do not increase branch coverage. Suppose a DB-state $state_1$ is already executed by $\mathcal{T}$ and we have a new candidate DB-state $state_2$. Let δ_{12} be the set of fields where $state_1$ and $state_2$ differ in their values. It is easy to see that $state_2$ can increase $\mathcal{T}$'s branch coverage over $state_1$ only if there exists a field in δ_{12} that is (1) retrieved by $\mathcal{T}$'s queries, and (2) used by an uncovered branch's condition. Intuitively, $state_2$ contains a value that can flow via $\mathcal{T}$ to affect an uncovered branch. Otherwise, the candidate $state_2$ can be discarded. Since the intersection of the fields retrieved by a test and the fields used by branches is usually small, most candidates can be discarded.

We use the above insight to design a filtering algorithm. The algorithm relies on branch analysis and branch-projected view computation, as described next.

1) Branch Analysis: DBGRILLER uses a combination of static and dynamic analysis to identify which tables and columns affect a branch condition.

Static Analysis. DBGRILLER uses static taint analysis to identify DB-dependent branches whose conditions are affected by values retrieved from the database (i.e., database is the source and branch conditions are the sinks), and which table and which field is involved in the branch condition.

Dynamic Analysis. Static analysis may not be precise enough to identify the exact table that a branch condition depends on. For example, in a language that supports polymorphism, static analysis may identify an interface whose exact type can only be determined at runtime. Knowing the exact type is important because in Object-Relational Model, there exists a one-to-one mapping between a type and an underlying database table. For instance, developer can define that each `user` associates with an `account`, where the type of `account` is polymorphic [16], meaning that it can be a `FacebookAccount`, a `TwitterAccount` or any account type class defined in the application, where each class corresponds to a different database table. The actual account type will be decided dynamically, depending on the value of `account_type` field of the `User` object. Therefore if a branch condition involves an `account`, which table this branch depends on cannot be determined statically. To tackle this issue, DBGRILLER compliment static analysis with dynamic tracing to identify the type of each variable instead of static type inference. Specifically, DBGRILLER instruments application and test code to dynamically identify the exact types and their corresponding tables during execution.

We use the example in Listing 1 to illustrate the process. DBGRILLER first performs static analysis and figures out that variable `item` is used in the branch condition on line 14, and the `quantity` and `inventory_id` field of `item` is used. Then it instruments and runs the test and finds that the instance of

Algorithm 1 Computing branch-projected views

```
1:  procedure BranchProjectedView(test, dbState, branches)
2:      queries ← FilterAndProjectQuery(test, branches)
3:      try
4:          view ← run queries with dbState in memory
5:          return view
6:      catch QueryFailExpection e
7:          Populate database with dbState
8:          view ← run queries against the database
9:          return view
10:     end try

11: procedure FilterAndProjectQuery(test, branches)
12:     queries ← All queries in test
13:     output_queries ← ∅
14:     for q ∈ queries do
15:         used_in_branch ← False
16:         for table t involved in q do
17:             if t is not used in any branch in branches then
18:                 continue
19:             else if some branch in branches depends on t but not on
        any specific field of t then
20:                 used_in_branch ← True
21:             else
22:                 fields ← all fields of t involved in branches
23:                 q← replace SELECT t.* with SELECT fields in q
24:                 used_in_branch ← True
25:         if used_in_branch == True then
26:             output_queries.add(q)
27:     return output_queries
```

item used in the branch condition is of type LineItem which derives from ActiveRecord class and maps to line_items table. Combining static and dynamic analysis, DBGRILLER understands that the branch condition on line 14 involves the quantity,inventory_id field of the line_items table (as well as inventories table by a similar analysis).

2) Computing Branch-Projected View: At the core of DBGRILLER's filtering algorithm is computation of *branch-projected views* that enables DBGRILLER to efficiently identify and discard DB-states that do not increase branch coverage. Intuitively, a branch-projected view of a DB-state is a set of its tables/fields that are retrieved by the test $\mathcal{T}$ and can potentially influence the execution of an uncovered branch. We define a view $view(T, S)$ of a test T and a DB-state S to be the results of all queries executed by T on the DB-state S. Based on the branch analysis results, not all tables/fields in a view may affect a set of given branches B. A projection Π_B of a view keeps only the columns that affect any of the branches B. Thus, for a test T and the set of uncovered branches B, if two DB-states S and S' have the same branch-projected views, i.e., $\Pi_B(view(T, S)) = \Pi_B(view(T, S'))$, they affect branches in B in the same way. Hence if S is already executed by $\mathcal{T}$, S' can be discarded as duplicate or equivalent.

DBGRILLER computes branch-projected views in two steps, as shown in Algorithm 1. First, it rewrites the queries made by the given test such that the rewritten queries contain only the tables/fields that affect uncovered branches (procedure FILTERANDPROJECTQUERY). Given a test and a set of branches to cover, it identifies all queries executed by the test (by examining query logs of the underlying database) (Line 12). It ignores the queries that involve only tables/fields that none of the branches depend on (Line 18). Remaining queries are included in the output. However, if all uncovered branches depend only on a subset of fields in a query in the output, it is rewritten to select only those fields (Line 23).

Next, DBGRILLER executes the rewritten queries (procedure BRANCHPROJECTEDVIEW). However, running a large number of such queries on a database can be very time-consuming, taking up to tens of seconds due to the cost to populate database and expensive database interaction. To reduce the cost, DBGRILLER loads the DB-state in memory (e.g., into a DataFrame in Python) and tries to run all the queries in-memory through a lightweight interface [13] (Line 4). While this reduces the view-construction time (to half a second in our experiments) and works for the majority of the queries, a few queries cannot be processed due to limited SQL syntax support by the in-memory processing engine. In this case DBGRILLER will fall back to populating the database with DB-state and running the queries against it (Line 7-8). The query results represent the branch-projected view for the given test, DB-state, and branches.

3) Discarding unhelpful candidates: Finally, DBGRILLER uses branch-projected views to compare DB-states: A newly generated DB-state S is unhelpful in increasing branch coverage if there exists an already selected state S' such that they both have the same branch-projected view computed with respect to the currently uncovered branches.

Algorithm 2 illustrates the process of filtering candidate states for a given test. To initialize for the test, it computes its branch-projected view over its default DB-state and all the DB-dependent branches in the application (Line 2-4). This view is saved to global state prior_view.

For each new candidate DB-state dbState for the test, DBGRILLER invokes the procedure FILTERCANDIDATE. It first retrieves the set of branches that are not yet covered by the test, and discard dbState if no such uncovered branches exist (Line 7-9). Then it computes the branch-projected view view of dbState and performs the important step of determining if the dbState is unhelpful. This is done by checking if view matches any already selected views, stored in prior_views. Note that the set of uncovered branches strictly shrinks, and the views in prior_views may be computed over a superset of branches projected in the current view. Therefore, the comparison with prior_views must be made after their projections with respect to the tables/columns that affect current set of branches. This is shown by the $\Pi_{branches}$ operator in Line 11. If the comparison finds a match, dbState is discarded. Otherwise, view is added to prior_views and dbState is passed to the constraint checker. After the checker confirms its validity , the test is invoked with the dbState (Line 17).

During execution of the test with a candidate state, DBGRILLER monitors if any new branch is covered; and if so, it is removed from the global set of uncovered branches (Line 18-19) used by future invocations of FILTERCANDIDATE. This feedback ensures that the set of uncovered branches shrinks over time. This has two important implications. First, it ensures the correctness of our comparison based on projection over currently uncovered branches (Line 11). Second, as the set

Algorithm 2 Filtering Candidate DB-State Using View

```
1: procedure INIT(test)
2:     state_T ← default db-state of test as defined by developer
3:     branches ← All branches in the application
4:     view ← BRANCHPROJECTEDVIEW(test, state_T, branches)
5:     prior_views ← { view }
6: procedure FILTERCANDIDATE(dbState,test, prior_views)
7:     branches ← Set of branches uncovered by test
8:     if BRANCHES is empty then
9:         Discard state and return
10:    view ← BRANCHPROJECTEDVIEW(test, dbState, branches)
11:    is_dup ← True, if there exists a v ∈ prior_views such that Π_branches(v) = view, False otherwise
12:    if is_dup == True then
13:        Discard candidate
14:    else
15:        prior_view.add(view)
16:        if state passes constraint checker then
17:            run test on state
18:            if new DBdependent branch b covered then
19:                Remove b from branches
```

of branches shrinks, the branch-projected views contain fewer tables/columns, making it more likely for a new candidate state to match an already selected state in Line 11. Therefore, over time, more and more candidate states are discarded, avoiding the expensive invocations of checker and test.

C. Optimizations

We add several optimizations that leverage branch analysis to avoid generating DB-states that will eventually be discarded. First, when mutating fields, instead of mutating every field, we only mutate the ones that are used in branch conditions, as the remaining fields will not be included in the view. Second, we only delete tuples from tables involved in branch conditions for a similar reason. Third, we compute hash for each state and perform duplicate checking, only keeping distinct DB-states as some mutations may produce already-seen states.

VII. EVALUATION

A. Experiment setup

We evaluate DBGRILLER on the 9 open-source web applications listed in Table I. Note that, the test suites provided by web developers do not necessarily touch all web pages. Our evaluation focuses on those web pages that are touched by existing test suites. Our prototype of DBGRILLER is built in Python and Ruby. Ruby code performs static analysis on the application source code while the Python code handles the rest of the workflow. We run DBGRILLER on a server with a 4-core 2.4GHz processor and 16GB memory. The branch coverage is counted using simplecov [17], which is the most popular library already used in 8 out these 9 applications to report coverage. We sequentially generate DB-states for each test, and set a time limit, 120h, to stop DBGRILLER if the limit is reached.

B. Branch Coverage Results

►How effective is DBGRILLER in increasing DB-dependent branch coverage? To answer this, we perform static analysis to identify all DB-dependent branches as described in §VI-B1. We then follow the standard branch coverage counting: every if-else branch condition that is DB-dependent presents two DB-dependent branches to cover, the `if` branch and the `else` branch.

TABLE II
CAPABILITY IN FLIPPING THE OUTCOME OF DB-DEPENDENT BRANCH CONDITION.

	Fr	Lb	Ch	Sp	Tr	Hg	Os	Al	Gs	**Avg**
ifelse-partial	409	53	71	208	143	98	170	95	55	**145**
ifelse-fully	142	20	37	84	52	16	44	29	22	**49**
%	35%	38%	52%	40%	36%	16%	26%	31%	40%	**35%**

Figure 2 shows the branch coverage of the tests in different applications with existing tests (provided by developer) and with the extra DB-states produced by DBGRILLER. As shown, DBGRILLER improves the DB-dependent branch coverage by 14 percentage points on average (up to 31 percentage points), increasing the overall DB-dependent branch coverage from 42–69% to 51–80%.

We further look into the capability of DBGRILLER in flipping the outcomes of DB-dependent branch conditions. Intuitively, if a DB-dependent branch condition has never been evaluated by existing tests, covering the corresponding `if` and `else` branches may require test input changes in addition to database state changes, which is beyond the scope of DBGRILLER. Instead, if a DB-dependent branch condition has been evaluated by existing tests and yet its outcome has always led to one branch (e.g., the `if`), additionally covering the other branch (e.g., `else`) could be achieved through a new database state. We refer to these cases as *turning partially covered if-else branch pairs into fully covered*. As shown in Table II, on average, 35% (up to 52%) of the partially-covered if-else branch pairs become fully-covered under DBGRILLER, showing that DBGRILLER is effective in flipping the outcome of DB-dependent branch conditions. For instance, the existing test on order checkout only covers the `if` branch shown in Listing 1 Line 14 where the checkout can complete, but not the `else` branch (Line 16). DBGRILLER successfully generated new database states to help cover the `else` branch, effectively checking how the application performs under insufficient inventory.

As DBGRILLER generates DB-states for each test individually, we also count the per-test coverage, with the average number of DB-dependent branches covered originally and after running DBGRILLER shown in Table III. With only a few number of DB-state to run per test (43 DB-states on average, as we will show in §VII-C), these DB-states are able to significantly increase the number of DB-dependent branches covered per test (33% on average), showing the capability of DBGRILLER to explore diverse test behavior for each individual test.

►How much does each mutation mechanism contribute to the increased coverage? To answer this, we configure DBGRILLER to use only one mechanism at a time, and report its coverage increase compared to the overall increase of DBGRILLER (100%). Figure 3 shows the result. We can

TABLE III
THE NUMBER OF COVERED DB-DEPENDENT BRANCH AVERAGED *per-test* (ORIGINALLY COVERED BY EXISTING TESTS AND AFTER RUNNING DBGRILLER), AND THE % OF INCREASE BY DBGRILLER.

	Fr	Lb	Ch	Sp	Tr	Hg	Os	Al	Gs	**Avg**
orig	20.1	17.2	4.3	30.3	20.7	8.5	11.2	7.7	13.3	**14.8**
DBGRILLER	29.3	20.1	6.2	37	27.3	10.2	16.6	11.6	15.7	**19.3**
% increase	46%	17%	44%	22%	32%	20%	48%	51%	18%	**33%**
w/o other tests	22%	10%	38%	12%	24%	6%	21%	41%	11%	**21%**

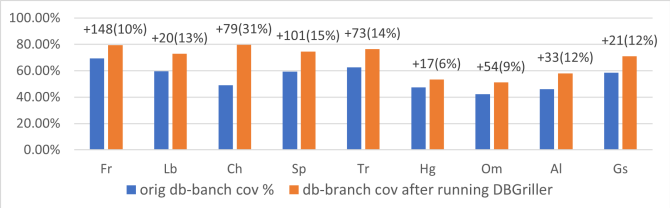

Fig. 2. DB-dependent branch coverage in %, with the number and percentage of branch increase at the top of orange bar.

see that among all the mutation mechanisms, MutateField and DeleteField are often the most effective methods. Note that these mechanisms sometimes overlap where some branches covered by DB-states generated with one mechanism are also covered using another mechanism. However, none of these mechanisms alone can achieve the same increase as overall, showing that they are still complimentary to each other.

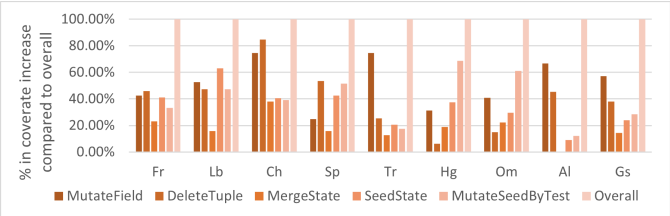

Fig. 3. DB-dependent branch coverage in %

►How much does DBGRILLER rely on DB-states from other tests? For a given test $\mathcal{T}$, DBGRILLER leverages not only its own DB-state, but also DB-states from other tests. In the worst case when no other tests are available, DBGRILLER can still produce new DB-states for $\mathcal{T}$ by mutating its own DB-state, thanks to its MutateField and DeleteTuple mutation mechanisms. The last line in Table III shows this worst case scenario: DBGRILLER is still able to achieve an average coverage increase of 21%.

C. Efficiency and effectiveness Results

DBGRILLER's efficiency comes from its ability to quickly filter out many unhelpful DB-states. Its effectiveness comes from its ability to produce mostly valid DB-states. We now empirically evaluate these two aspects, with various statistics shown in Table IV.

►Is tool effective in discarding unhelpful DB-states? As shown in Table IV, various mutation mechanisms of DBGRILLER generate a large number of DB-states (up to 303K). This number varies across applications because their numbers of seed database differ. Despite the large number of initial candidates, DBGRILLER is able to filter out most of them, from 78% up to 97%, leaving only a few thousands for validation (shown in highlighted row **% reduced in filter**), showing the effectiveness of DBGRILLER's candidate filtering algorithm. Effective filtering is crucial for tool's efficiency because running the checker on all generated DB-states without filtering and running tests on all valid DB-states would be prohibitively expensive (a few months for the applications in Table IV) while still producing the same branch coverage.

►Is tool effective in generating valid and many DB-states? The **% invalid candidate** row in Table IV shows that only a small fraction of DB-states after filtering is discarded by the checker (2-10%). This highlights that the DB-states produced by DBGRILLER's mutation mechanisms are mostly valid. The **DB-states** row shows that the final number of unique DB-states that pass the checker is large (0.6K-34.9K), which achieves the coverage increase reported in Figure 2. This highlights that DBGRILLER, despite filtering a large number of unhelpful DB-states, is able to produce a large number of diverse and valid DB-states.

►How much time does DBGRILLER take? The **overall time** row in Table IV shows the total time spent to *sequentially* run our experiments for each application. The time includes the time to generate DB-states, filtering them, validating them, and running tests on them. Three applications, *Forem*, *Spree* and *OpenStreetMap* include many tests (e.g., it takes over 8h to run all original tests in *Forem*), and DBGRILLER could not finish running all the tests within our time limit of 120h. The remaining applications take from 8h to 66h. We believe the overhead is acceptable for offline testing. It can be potentially accelerated by running multiple tests in parallel or in a continuous integration test environment that runs only the tests that are affected by recent code changes.

Figure 4 shows the breakdown of total time spent into state generation+filtering, validation, and test execution time. Despite generating a large number of DB-states, generation+filtering step is the fastest in all but one applications (*Forem*). The majority of the time is still spent on DB-state validation and running tests. This is because filtering is almost an order of magnitude cheaper than the other two steps (row **generation + filtering time** in Table IV). In comparison, the time it takes to validate a DB-state or to run a test is much longer, from a few seconds to tens of seconds.

DBGRILLER's filtering efficiency comes from the fact that even though it requires creating views by running queries, DBGRILLER is able to run most of the queries in-memory (Line 4 in Algorithm 1) without interacting with the actual database. This happens for 81% to 100% of the queries, as shown in the *% views constructed in-mem* row in Table IV.

D. Comparison with Random Fuzzing

We compare DBGRILLER with simple baselines that generate random DB-states using the fuzzing tool Zest [39]. Because all the applications we tested are written in Ruby while Zest

TABLE IV
NUMBER OF DB-STATES PRODUCED BY DBGRILLER, RUNNING TIME BREAKDOWN, AND OVERALL RUNNING TIME AND STATE COUNT.

		Fr	Lb	Ch	Sp	Tr	Hg	Os	Al	Gs	Avg
# candidate	# before filter	303K	7.6K	114K	217K	77.2K	50.1K	71.4K	13.4K	33.6K	9.9K
	% reduced in filter	**97%**	**91%**	**91%**	**82%**	**84%**	**94%**	**78%**	**91%**	**87%**	**88%**
DB-state	# after filter	8.4K	0.7K	10.1K	39.0K	12.1K	3.0K	15.9K	1.4K	4.3K	10.5K
	% invalid candidate	**2%**	**7%**	**4%**	**10%**	**10%**	**5%**	**10%**	**5%**	**9%**	**7%**
	# candidates pass checker	8.3k	0.6K	9.7K	34.9K	11.0K	2.9K	14.3K	1.3K	3.9K	9.6K
running time breakdown per-test	% views computed in-memory	99%	90%	91%	96%	99%	100%	81%	100%	96%	95%
	generation + filtering time (per-candidate)	0.7s	2.9s	1.0s	0.6s	0.3s	0.1s	4.1s	0.3s	1.0s	1.3s
	validation time (per-candidate)	13.1s	11.4s	7.1s	2.2s	8.3s	6.2s	6.4s	34.5s	13.5s	11.4s
	test time (per-test)	13.9s	28.4s	9.4s	8.3s	10.7s	9.8s	20.2s	24.7s	17.6s	15.9s
Overall	# target test processed	120	67	374	378	232	155	160	103	144	193
	# DB-states	**8.3K**	**0.6K**	**9.7K**	**34.9K**	**11.0K**	**2.9K**	**4.3K**	**1.3K**	**3.9K**	**9.6K**
	# states per test	65	9	26	92	47	19	90	13	27	43
	time per test	57min	7min	9min	19min	17min	6min	45min	14min	16min	21min
	overall time running original tests	8h	0.6h	1.1h	1.1h	0.8h	0.5h	3h	0.8h	1h	2h
	overall time w/ DBGRILLER (sequential)	120h	8h	56h	120h	66h	15h	120h	24h	40h	63h

Fig. 4. The breakdown of time spent in filtering candidate, validating candidate DB-state and running test on DB-state.

works on Java programs, we only use the generator-based fuzzing framework but not the coverage-guided feature. We write a generator that takes in a database schema and randomly generates values for each table field based on two strategies we describe next. We also implement the `Assume` function that calls the constraint checker to check the validity of the generated DB-state, leaving the test body empty.

For generating values for each field, we use two strategies:

Random values: This strategy generates random values for each table field according to its type (integer, float, string, etc.). Each generated table contains a random number of tuples ranging from 0 to 10 (a larger number causes Zest to crash). For each application, we set the time budget as the amount of time DBGRILLER takes (shown in Table IV).

The number of DB-states produced and those that passed the validation is shown in Table V. Even though the baselines produced a large number of DB-states, none of them passed the validation. This is because randomly-generated field value easily fails even simple single-field constraint, like a string field which expects JSON format, or an integer field storing a ratio from 0 to 100. *Random* easily fails because any failed constraint would make the database invalid and there are hundreds of constraints per application.

Seed values: Based on the above observation, we improve the baseline by using only values from seed databases. Instead of generating a random value for each field and a random number of tuples for each table, this strategy randomly samples a value of the same table field from seed databases, and sets the bound for the number of tuples in a table to the maximum number of tuples in the same table in seed databases (usually smaller than 10). We generate the same number of DB-states as *Random*.

TABLE V
NUMBER OF STATES GENERATED/PASS-CHECKER BY FUZZING BASELINE.

	Fr	Lb	Ch	Sp	Tr	Hg	Os	Al	Gs
DB-states	20.0K	1.9K	22.7K	16.8K	18.8K	6.7K	45.3K	2.2K	9.3K
Random # pass check	0	0	0	0	0	0	0	0	0
Seed # pass check	0	0	0	0	1.9K	5.2K	0	0	0
increased cov	0	0	0	0	15	18	0	0	0

This improved strategy improves the situation slightly: it succeeded in generating valid databases for two of the applications, *Tracks* and *Huginn*. These are the two applications with the smallest number of tables, fields and constraints among all applications, yet random fuzzing still has a much lower chance to produce a valid state compared to DBGRILLER. We run all the tests on the valid databases and report the DB-dependent branch coverage increase in Table V. Compared to DBGRILLER, random fuzzing achieves a slightly better increase (+18 compared to +17) for *Huginn* and a much lower increase (+15 compared to +73) for *Tracks*. For the remaining 7 applications, DB-states generated by this strategy are often failing on constraints involving multiple fields or tuples, like unique constraint, foreign key constraint, functional-dependency (e.g., the value of two fields must be equal), etc.

E. Analysis of Uncovered Branches

To understand DBGRILLER's potential limitations in increasing branch coverage, we examine two applications with relatively small coverage increase: *Huginn* and *OpenStreetMap*. Many of *Huginn*'s DB-dependent branches are related to parsing the JSON text read from a database field. As DBGRILLER mostly uses seed values or random values and *Huginn*'s seed value does not include diverse JSON text, DBGRILLER is unable to cover these branches. For *OpenStreetMap*, a test often includes multiple assertions (instead of one at the end of the test, like tests in other apps) and when the first assertion fails (as DBGRILLER changes DB-state, test-specific assertions often fail), the test returns without executing the code after the assertion, limiting coverage. This can be potentially addressed by removing/disabling test assertions when using DBGRILLER-produced DB-state.

TABLE VI
BUGS FOUND AND FP RATE OF DBGRILLER AND CODEQL

Method	Fr	Lb	Ch	Sp	Tr	Hg	Os	Al	Gs	FP-rate
DBGRILLER-found	11	2	0	5	2	4	0	0	2	15%
DBGRILLER-real	9	2	0	3	2	4	0	0	2	-
codeql-found	60	16	35	26	6	2	23	24	34	97%
codeql-real	2	1	0	0	0	0	0	1	2	-

Then we sample 60 uncovered branches and look into why DBGRILLER fails to cover them. We find three main reasons. First, 55% of uncovered branches require specific values of one or multiple fields, or the result of an aggregate query, whose values are not included in the seed databases. Second, 28% of branches are affected not only by the database but also with other inputs such as user input and application configuration. Changing DB-states alone is not sufficient to cover them. Third, 17% of branches are unreachable for any valid DB-state. Examples include a branch checking a field value which is being reset to a constant before the branch.

F. Bugs found

Like all techniques that increase testing coverage[28], [37], DBGRILLER can help testing to expose more bugs by exercising more and important code paths. However, DBGRILLER is *not* a bug-detection tool by itself. Good testing oracles have to be in place to report the manifestation of bugs. In the following, we offer a baseline evidence of this with the most generic and basic type of oracle that reports a likely bug when Rails catches an unhandled exception (and eventually returns the HTTP error `404: Not Found`). We expect many more bugs to be detected with more application-specific oracles.

With this basic oracle, we found 34 distinct cases of unhandled exceptions from 6 applications. Manual inspection showed that 22 are caused by actual bugs in the applications, 8 are caused by incomplete test cases (exceptions unhandled by tests but are actually handled in the application), and 4 are false positives — the test database violates data constraints that DBGRILLER fails to extract due to the limitations discussed in SectionVIII. Among the actual bugs, 12 are confirmed by developers and 3 are already fixed.

Most of these bugs are null bugs, a common bug in database backed applications [19] which directly invokes method from a query result without checking whether it is null or not. In retrospect, these bugs look simple, but are actually hard to accurately pinpoint. To demonstrate this, we use a static null-checker. We are not aware of any publicly-available static null-checker for Ruby-On-Rails applications, and hence we build our own based on CodeQL [4], [43]. The checker analyzes dataflow (provided by CodeQL) and reports a potential null bug if (1) there is a database query result object on which some method is invoked later, and (2) between where the query result is computed and where a method is invoked on it, there is no branch condition inspecting if the query result is null.

Table VI compares the static checker with DBGRILLER. The checker has a high false positive rate of 97%, and it finds only 6 real bugs, much fewer compared to DBGRILLER. This is due to multiple factors including (1) fundamental limitations of static analysis of a dynamic language such as Ruby [35], (2) widely-used asynchronous callbacks [14] that are hard to analyze statically [44] and are not analyzed by CodeQL that we use, (3) static analysis not being aware of data constraints implicitly imposed by database and validation functions (e.g., certain query result will never be NULL), and so on.

VIII. DISCUSSION

Generality. Although DBGRILLER's prototype is built for Rails applications, none of its components are Rails specific. Some features DBGRILLER relies on, e.g., `Model`-like class to map object class to tables, data constraints specified through validation callback, are common in other ORM frameworks (model class and validation callback in Django[6], persistent class and bean validator in Hibernate [9], etc.).

Limitations. DBGRILLER's constraint extraction and DB-related branch analysis have limitations due to static analysis. The constraints extracted by DBGRILLER are sound but incomplete, as it does not detect constraints implied by *h*ow the value in database is generated (e.g., the `score` field of a post is computed from field `votes`, but DBGRILLER cannot detect such constraint). The static analysis on branches is conservative and may include more fields and tables than actually used, making DBGRILLER select more candidates than necessary. Dynamic taint analysis is potentially more accurate, but unfortunately not supported in Ruby right now.

IX. RELATED WORK

Testing database-backed applications. Prior works apply symbolic execution related techniques to test database-backed applications. SynDB [41], [40] replaces the interaction with a database system with a synthesized interaction that runs queries on a symbolic database, then applies dynamic symbolic execution. It considers all program, query and database constraints in tandem and generates both test input and database state. XDataPro [22] performs static analysis on the program code to collect the conditions that a database state needs to satisfy in order to follow a certain program path, and uses a solver to generate state. However, these tools model a small subset of SQL semantics and are only good for small to medium size, self-contained applications, while DBGRILLER is targeting real-world applications which issue arbitrary query and use external libraries.

Testing big data applications. Much research has explored testing big data applications. Prior work widely explored testing Spark applications, which is composed of calls of Spark functions and user-defined functions (UDFs) and works on a database state. BigTest [23] uses symbolic execution to automatically enumerate different path conditions and generate database states using an SMT solver. BigFuzz [47] proposes an efficient fuzzing technique that focuses on exploring paths in user-defined functions instead of Spark libraries. Big data applications differ from database applications in three major aspects. 1) Big data applications targeted in prior work are expected to work on any database state. Consequently, data validity is not a concern. 2) The space of database states is

much smaller as they include only a few tables. 3) The Spark libraries only include a few data operators which are relatively easy to model, while modeling complete SQL semantics requires huge effort. Furthermore, the complexity of UDFs in Spark program is much lower compared to the entire database application. As a result, these techniques cannot be directly applied to database applications.

Another line of work in testing big data applications targets performance testing of query workload. QAGen [27] takes in database schema, cardinality and data distribution constraints, MyBenchmark [34] takes in parameterized queries with derives constraints, and Arasu et. al. [24] proposed a language to specify data distribution constraints. They output large database states satisfying these constraints. Our work focuses on integrity constraints instead of distribution constraints.

Database testing. A lot of research work has explored generating queries to test database systems. SQLancer [21], [42] includes a series of techniques, including query partitioning and pivot query synthesis, to generate SQL queries paired with certain properties of query results. These queries can find logical bugs of database systems. AMOEBA [33] detects query performance bug by exploring equivalent query rewrites and check whether rewritten queries have similar performance. These techniques tackle the validity challenge by designing program mutations to produce queries with desired properties such semantic equivalence to a given query. In contrast, DBGRILLER's mutation produces states that are likely-valid only empirically because it would be impossible to generate guaranteed valid states due to the complexity of application-specific data constraints.

Random testing and fuzz testing. A big challenge for random and fuzz testing is to generate valid inputs. Randoop [36] permutes method-call sequences to generate valid inputs, and eliminates redundant execution by keeping track of the sequences. Zest [39], [38] proposes generator-based testing that increases the chance to produce a semantically valid input by converting random-input generators into deterministic parametric generators. Other work proposed grammar-based fuzzing techniques. Saffron [32] relies on the user to provide an approximate grammar and refines it during the fuzzing process. Superion [45] proposes grammar-aware grey-box fuzzing that mutates the abstract syntax tree of the test input instead of random mutation to increase code coverage. Such techniques can be potentially used for database applications, yet designing application-specific grammar or DB-state generator can be very challenging due to the large number of data constraints.

Testability transformation (TT). TT [30] transforms programs to improve the performance and effectiveness of test data generation techniques. Previous works proposed various TT techniques, e.g., to enrich functions to return better descriptive and heuristics values that can enhance search-based software testing [25], to remove from programs binary flags that can hurt evolutionary testing[29], [26], to produce simplified and coverage-preserving versions of a program that are easier to analyze/test and for which test data generation is easier [29], [31]. DBGRILLER introduces a novel kind of TT by transforming parts of an application's logic into a DB-state validity checker, which helps to ensure that a generated DB-state is reasonable for tests.

X. CONCLUSION

Generating test databases for database-backed applications is challenging, due to the large space and validity of the databases, and the long running time to validate and run tests. In this paper we propose DBGRILLER to address these challenges. DBGRILLER 1) extracts application constraints into a checker program to validate any given database state, and 2) leverages seed databases designed by developers and computes branch-projected views to generate likely-valid DB-states that are prone to increase branch coverage. Evaluation shows its effectiveness in increasing test coverage and exposing bugs in database-backed applications.

Acknowledgments. We would like to thank the anonymous reviewers for their insightful comments on the paper. Shan Lu's research is partly supported by NSF (grants CCF-2119184, CNS-1764039), the CERES Center for Unstoppable Computing, and the Marian and Stuart Rice Research Award.

REFERENCES

[1] *American Fuzz Loop*. http://lcamtuf.coredump.cx/afl/.
[2] *Autolab: course management service that enables auto-graded programming assignments.* https://github.com/autolab/Autolab.
[3] *Chatwoot: open-source customer engagement suite, an alternative to Intercom, Zendesk, Salesforce Service Cloud etc.* https://github.com/chatwoot/chatwoot.
[4] *CodeQL: the analysis engine used by developers to automate security checks, and by security researchers to perform variant analysis*. https://codeql.github.com/docs/.
[5] *Diaspora: A privacy-aware, distributed, open source social network.* https://github.com/diaspora/diaspora.
[6] *Django, a python web application framework*. https://www.djangoproject.com/.
[7] *Forem: open source software for building communities*. https://github.com/forem/forem.
[8] *Growstuf: open data project for small-scale food growers*. https://github.com/Growstuff/growstuff.
[9] *Hibernate, an ORM framework for java*. http://hibernate.org/orm/.
[10] Huginn: create agents that monitor and act on your behalf. https://github.com/huginn/huginn.
[11] *Lobsters: computing-focused community centered around link aggregation and discussion*. https://github.com/lobsters/lobsters.
[12] *Openstreetmap: rails application powering OpenStreetMap*. https://github.com/openstreetmap/openstreetmap-website.
[13] *Pandas: read SQL query or database table into a DataFrame*. https://pandas.pydata.org/docs/reference/api/pandas.read_sql.html.
[14] *Rails ActiveRecord callback mechanism*. https://guides.rubyonrails.org/active_record_callbacks.html.
[15] *Rails model APIs*. https://devhints.io/rails-models.
[16] *Rails polymorphic association*. https://guides.rubyonrails.org/association_basics.html#polymorphic-associations.
[17] *SimpleCov: code coverage tool for ruby*. https://github.com/simplecov-ruby/simplecov.
[18] *Spree: open source headless multi-language/multi-currency/multi-store eCommerce platform*. https://github.com/spree/spree.
[19] *Top errors from Ruby on Rails project*. https://rollbar.com/blog/top-10-errors-from-1000-ruby-on-rails-projects-and-how-to-avoid-them/.
[20] *Tracks: web-based application to help you implement David Allen's Getting Things Done methodology*. https://github.com/TracksApp/tracks.
[21] Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, Banff, Alberta, November 2020. USENIX Association.

[22] Pooja Agrawal, Bikash Chandra, K. Venkatesh Emani, Neha Garg, and S. Sudarshan. Test data generation for database applications. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1621–1624, 2018.

[23] Muhammad Ali Gulzar, Madanlal Musuvathi, and Miryung Kim. Bigtest: A symbolic execution based systematic test generation tool for apache spark. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 61–64, 2020.

[24] Arvind Arasu, Raghav Kaushik, and Jian Li. Data generation using declarative constraints. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, page 685–696, 2011.

[25] Andrea Arcuri and Juan P Galeotti. Testability transformations for existing apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 153–163. IEEE, 2020.

[26] David W Binkley, Mark Harman, and Kiran Lakhotia. Flagremover: A testability transformation for transforming loop-assigned flags. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(3):1–33, 2011.

[27] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. Qagen: Generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, page 341–352, 2007.

[28] Ankit Choudhary, Shan Lu, and Michael Pradel. In *ICSE*, pages 266–277, 2017.

[29] Mark Harman, André Baresel, David Binkley, Robert Hierons, Lin Hu, Bogdan Korel, Phil McMinn, and Marc Roper. Testability transformation–program transformation to improve testability. In *Formal methods and testing*, pages 320–344. Springer, 2008.

[30] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[31] Robert M Hierons, Mark Harman, and CJ Fox. Branch-coverage testability transformation for unstructured programs. *The Computer Journal*, 48(4):421–436, 2005.

[32] Xuan-Bach D. Le, Corina Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. Saffron: Adaptive grammar-based fuzzing for worst-case analysis. *SIGSOFT Softw. Eng. Notes*, 44(4):14, sep 2021.

[33] Xinyu Liu, Qi Zhou, Joy Arulra, and Alessandro Orso. automatic detection of performance bugs in database systems using equivalent queries. In *ICSE*, 2022.

[34] Eric Lo, Nick Cheng, and Wing-Kai Hon. Generating databases for query workloads. *Proc. VLDB Endow.*, 3(1–2):848–859, sep 2010.

[35] Magnus Madsen. Static analysis of dynamic languages. *: http://pure. au. dk/ws/files/85299449/Thesis. pdf*, 2015.

[36] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE*, 2007.

[37] Rohan Padhye, Caroline Lemieux, and Koushik Sen. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 398–401, 2019.

[38] Rohan Padhye, Caroline Lemieux, and Koushik Sen. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 398–401, 2019.

[39] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. *Semantic Fuzzing with Zest*. 2019.

[40] Kai Pan, Xintao Wu, and Tao Xie. Automatic test generation for mutation testing on database applications. In *ASE*, 2013.

[41] Kai Pan, Xintao Wu, and Tao Xie. Guided test generation for database applications via synthesized database interactions. *ACM Trans. Softw. Eng. Methodol.*, 23(2), apr 2014.

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

[43] Max Schäfer and Oege de Moor. Type inference for datalog with complex type hierarchies. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, page 145–156, 2010.

[44] Thodoris Sotiropoulos and Benjamin Livshits. Static analysis for asynchronous javascript programs. *arXiv preprint arXiv:1901.03575*, 2019.

[45] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 724–735. IEEE Press, 2019.

[46] Junwen Yang, Utsav Sethi, Cong Yan, Alvin Cheung, and Shan Lu. Managing data constraints in database-backed web applications. In *ICSE*, page 1098–1109, 2020.

[47] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction. In *ASE*, 2020.