Protecting Data Integrity of Web Applications with Database Constraints Inferred from Application Code

Haochen Huang
hhuang@ucsd.edu
University of California, San Diego
USA

Bingyu Shen
byshen@eng.ucsd.edu
University of California, San Diego
USA

Li Zhong
lizhong@ucsd.edu
University of California, San Diego
USA

Yuanyuan Zhou
yyzhou@eng.ucsd.edu
University of California, San Diego
USA

ABSTRACT

Database-backed web applications persist a large amount of production data and have high requirements for integrity. To protect data integrity against application code bugs and operator mistakes, most RDBMSes allow application developers to specify various types of integrity constraints. Unfortunately, applications (e.g., e-commerce web apps) often do not take full advantage of this capability and miss specifying many database constraints, resulting in many severe consequences, such as crashing the order placement page and corrupting the store inventory data.

In this paper, we focus on the problem of missing database constraints in web applications. We first study several widely used open-source e-commerce and communication applications, and observe that all these applications have missed integrity constraints and many were added later as afterthoughts after issues occurred.

Motivated by our observations, we build a tool called CFinder to automatically infer missing database constraints from application source code by cleverly leveraging the observation that many source code patterns usually imply certain data integrity constraints. By analyzing application source code automatically, CFinder can extract such constraints and check against their database schemas to detect missing ones. We evaluate CFinder with eight widely-deployed web applications, including one commercial company with millions of users. Overall, our tool identifies 210 previously unknown missing constraints. We have reported 92 of them to the developers of these applications, so far 75 are confirmed. Our tool achieves a precision of 78% and a recall of 79%.

CCS CONCEPTS

• **Software and its engineering** → **Software verification and validation**; **Software reliability**.

KEYWORDS

Data integrity, Database constraints, Web applications, Static analysis

ACM Reference Format:
Haochen Huang, Bingyu Shen, Li Zhong, and Yuanyuan Zhou. 2023. Protecting Data Integrity of Web Applications with Database Constraints Inferred from Application Code. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3575693.3575699

1 INTRODUCTION

1.1 Problem: Missing Database Constraints

Data integrity is critical for database-backed web applications used in e-commerce, banking, and many aspects of our daily life [27]. As various data redundancy techniques have been used in computer storage and network subsystems, integrity issues caused by hardware errors [6, 36, 61] or crash failures [8, 31] have been reasonably well addressed in today's data centers. In contrast, application bugs or operator errors are significantly understudied and remain as increasingly pervasive root causes for data integrity issues in databases [27].

Fortunately, most of today's relational database management systems (RDBMS) provide integrity constraints that help applications to guarantee desired data integrity [9, 52]. Specifically, application developers specify database constraints based on their own business logic and enforce them in the database schema, such as a *not-null* constraint for order.total, or a *unique* constraint for user.email. Such database constraints would detect and refuse any incorrect data manipulation caused by either bugs in application code, or operator mistakes when directly manipulating data via the database administrator (DBA) console. Common constraints supported in popular RDBMSes include *Not-null*, *Unique*, and *Foreign key* constraints [38, 40, 43, 48].

The necessity of specifying database constraints to protect data integrity has received increasing attention. For example, central players in modern web frameworks, such as Rails (Ruby), Django (Python), and Hibernate (Java), have supported the migration helpers for all three common database constraints in recent years [10, 50, 62], enabling applications to easily specify and enforce constraints in databases.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03.
https://doi.org/10.1145/3575693.3575699

/*Saleor*/	/*Zulip*/	/*Django-oscar*/
Violated constraint: One *Order.total* field has ***null*** value.	**Violated constraint**: Two *UserProfile.email* are the same, not ***unique***.	**Violated constraint**: *Order.basket_id* is an integer-field rather than a ***foreign key*** to *Basket*.
Consequence: ***[Page crash]*** Shop admin cannot operate on the dashboard page as it crashes.	**Consequence**: ***[Block business logic]*** Block either user from logging in.	**Consequence**: ***[Data corruption]*** Potential data corruption and performance hits for user requests.
Resolution taken: Add ***Not-null*** constraint to **database**.	**Resolution taken**: Add ***Unique*** constraint to **database**.	**Resolution taken**: Add ***Foreign key*** constraint to **database**.
(a)	(b)	(c)

Figure 1: Three real-world issues [18, 54, 74] that violated three types of DB constraints and led to severe consequences. These issues are from popular open-source web applications. To fix the issues, the developers added the constraints to the database [16, 55, 73].

However, despite these initiatives, many app developers still do not take full advantage of database constraints to protect their application data integrity against application bugs or operator errors. We observe in our study (§ 2) that many constraints (10-72) for each studied app were added as *afterthoughts*, i.e., they were missed first when columns were created and added much later, often because a data integrity issue was detected and resulted in damages.

There are many reasons for such negligence. As the industry demands more engineers to build various applications, many of them do not have solid database training and may not be aware of data integrity or constraints at all [59, 60]; Additionally, to err is human, even experienced developers can easily forget some required constraints due to deadline pressure.

1.2 Consequences of Missing Constraints

Missing DB constraints can result in severe consequences. Figure 1 shows three real-world examples [18, 54, 74] from three widely-used e-commerce and team chat applications. These issues were caused by inconsistent data stored in databases that violate the *not-null*, *unique*, or *foreign key* constraint. As a result, the applications suffer from severe consequences, such as page crashes and failed login attempts. For e-commerce, any such issue can lead to significant business loss [54].

To fix the problem, and more importantly *to avoid similar issues in the future*, developers added the missing data constraints into their corresponding databases [16, 55, 73]. Had these constraints been specified earlier, such issues would have been detected and reported before invalid data being inserted into databases in the first place and avoid the impact on users.

Missing DB constraints has two primary consequences:

- Without database constraints to guard data integrity, data corruption caused by application bugs can easily stay dormant for a long time before being exposed, impacting users and leading to business loss. Without early detection, the culprit application bug can cause many corruptions, making database repairing a more challenging task.
- Missing database constraints also introduces challenges in diagnosing such issues because it is difficult to trace back and identify *when* and *how* such inconsistent or erroneous data were added into the databases.

We can look into one of the examples [54] in Figure 1(a) from the popular e-commerce Saleor [56]. The app developers noticed a page crash caused by "an invalid order in database with a null total price." However, they got stuck in identifying the root cause of the *null* record. After rounds of investigations in nine days, three developers finally found the application bug. To detect future similar application bugs earlier before they corrupt databases, developers added the *not-null* constraints into the database to "prevent reported weird and hard to reproduce bugs", according to their commit comment.

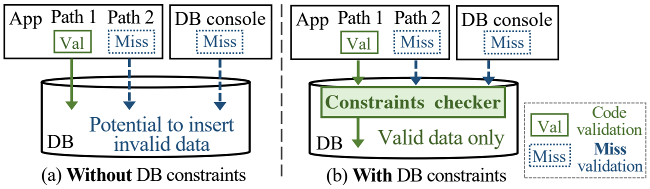


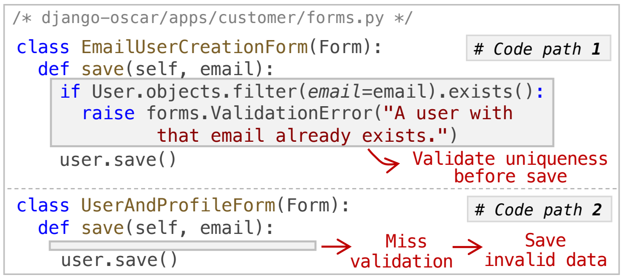
Figure 2: In (a) when the constraints are not enforced in DB, missing validation in any code path or DB console could potentially cause invalid data to be inserted. Contrarily, in (b), when DB constraints are enforced, even if validations are missed in some paths, DB always conducts integrity checks and blocks invalid data as the final guard.

1.3 Why DB Constraints Are Better Guards?

Interestingly, many developers think that their own application code can check against data integrity violations, and thereby there is no need to add DB constraints [7, 28]. Such assumptions often fail to protect data integrity in practice because there are multiple places that can change the database data and result in data integrity violations if not checked properly.

Specifically, as depicted in Figure 2, database data can be added or altered in various places throughout the application's code, and some of them may not even be in the same piece of software (e.g. some batch job scripts to insert or change data in bulk). To make things even worse, software may implement some code logic in a different language [37], or by different teams. The fast turnover rates of today's software engineers in IT companies further make it difficult to ensure that every single code location has proper integrity checks.

Figure 3 shows one such real-world example [17] from Oscar [44]. In this e-commerce application, each user's email field needs to be *unique* as it is used for authentication. To ensure this, when a new user signed up, the application code checked whether that email already existed in DB. Unfortunately, on another code path that performed email updating for registered users, there was no check at all. As a result, this application bug allowed the same



```
/* django-oscar/apps/customer/forms.py */
class EmailUserCreationForm(Form):                # Code path 1
  def save(self, email):
    if User.objects.filter(email=email).exists():
      raise forms.ValidationError("A user with
             that email already exists.")
    user.save()
-----------------------------------------------------------
class UserAndProfileForm(Form):                   # Code path 2
  def save(self, email):

    user.save()
```

Figure 3: A real-world issue [17] from Oscar caused by missing code validations. The user's email should be unique. Developers specified the check in one code path (when new users registered) but forgot to check in another code path (when a registered user updated his email). To fix it, they enforced the *unique* constraint in DB.

email addresses to be used for two or more user accounts, causing many login issues. It took developers quite some time to diagnose, and even much longer to repair the database (since they needed to inform the affected users to change to another email address).

Moreover, application code checks for data integrity often fail during concurrent executions because of data races [66]. For example, two concurrently handled requests can both receive non-existent results from the data integrity check in the first query, and then both insert the same values in the second query, which violates the unique constraint.

A study on Rails applications [4] reveals that 13% of code validations for uniqueness and foreign key are error-prone during concurrent executions, as is also warned in web frameworks' documentations [14, 51]. Our study in §2 also confirmed such observation. Even encapsulating validation logic within a transaction may not work because most production databases default to non-serializable isolation [4, 66].

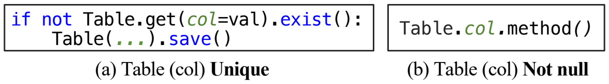
Furthermore, DB admins can also manipulate data using the "backdoor", i.e., the DB console, which bypasses all checks in the code. In comparison, in Figure 2(b), when constraints are enforced by the DB, even if validations are missed in some paths, the DB always acts as the final guard to perform integrity checks and detect violations against specified constraints.

As such, we believe that web applications should *take full advantage of database constraints to ensure data integrity* when possible.

1.4 Our Contributions

This paper focuses on the problem of missing database constraints in widely-used web applications that leads to data integrity issues and results in system downtime and business loss.

First, we make one of the first attempts in understanding and evaluating the reality of the adoption of database constraints in today's web applications. We study five popular web apps in Table 1 ranging from e-commerce to communication tools. Our study reveals several interesting findings: (1) Many (10-72) database constraints were missed in the beginning and were added much later as afterthoughts after some issues occurred. (2) Most (82%) of these cases could result in consequences, including page crash and data corruption of order-related or payment-related records. (3) Most (87%) issues that missed DB constraints also missed code checks



```
if not Table.get(col=val).exist():
   Table(...).save()
```

```
Table.col.method()
```

(a) Table (col) **Unique** (b) Table (col) **Not null**

Figure 4: Code snippets with implicit assumptions on database constraints. (a) Unique constraint: Save record only when no record filtered by the column exists. (b) Not-null constraint: Invoke methods on the column which should not be nullable.

in application code, indicating that solely relying on application code checks instead of leveraging database constraints is not a safe approach to guarantee data integrity (More details in §2).

Second, we leverage a unique observation that application code usually contains "hints" that imply certain data constraint assumptions made by developers. Figure 4 shows two examples of such code snippets. In (a), the code uses the column `col` as an identifier to check its existence, and only creates a new record if it does not already exist, indicating that the `col` is a `unique` identifier. In (b), the code invokes a method on `col`, indicating that `col` cannot be null. §3.3 shows all our discovered code patterns that imply data constraints.

By leveraging this observation, we build CFinder which employs program analysis to analyze application source code to automatically infer and detect any missing database constraints to improve database integrity (against application bugs and operator mistakes).

We evaluate CFinder with eight widely deployed web applications, including an industry-strength software from a **commercial company with millions of users**. CFinder has detected 210 missing DB constraints from these applications. We have reported **92 of them to the developers of these applications, so far 75 have been confirmed by these software.** The tool effectively detects the missing constraints with a precision of 78% for newly detected constraints and a recall of 79% for an existing dataset.

2 UNDERSTANDING MISSING DATABASE CONSTRAINTS IN WEB APPLICATIONS

Before we build a tool to infer the missing constraints, we first aim to understand more about the current status of DB constraints in web applications. Specifically, we aim to answer: (1) *Is it common for developers to miss specifying some DB constraints?* We define "*missing*" constraints as those that are not specified when the columns are created, and added *later* in another pull request. Missing constraints indicate the potential vulnerabilities which allow invalid data to get stored in the database. (2) *Do these missing DB constraints lead to issues with severe consequences?* Finally, (3) *Do these missing constraints have validation checks in the code and whether the validations can protect the data integrity effectively?*

As a lens to answer these questions, we conduct the study on five widely-deployed real-world web applications listed in Table 1, representing app domains including e-commerce, team chat, etc. The apps are built on top of Django [12], a popular framework powering more than 94K web apps, including large commercial companies like Instagram [23].

To collect the history of adding DB constraints, we leverage the database migration files [15], which maintain the historical modifications to the database schema. From them, we collect the

Table 1: The web applications used in our study. *Stars*: Number of stars on Github. *LoC*: Lines of code.

App.	Category	Stars	LoC	#Table	#Column
Oscar [44]	E-commerce	5.2K	74K	77	773
Saleor [56]	E-commerce	15.3K	298K	98	1013
Shuup [58]	E-commerce	1.8K	196K	227	2236
Zulip [79]	Team chat	15.3K	361K	97	826
Wagtail [63]	Content management	11.7K	181K	60	841

Table 2: The number of database constraints that are missed first and added in later pull requests in each application.

App.	**Oscar**	**Saleor**	**Shuup**	**Zulip**	**Wagtail**	**Total**
Unique	22	10	5	16	6	59
Not-null	48	9	6	9	4	76
Foreign key	2	2	0	4	0	8
Total	72	21	11	29	10	143

SQLs that add the new database constraints. To get the "*missing*" constraints, we further filter out the constraints that are added together with the creation of columns. To collect the related issues, we search the issue tickets that reference the commit of migration files. We then manually examine the issues to understand the root causes and severity based on developer comments and issue labels.

Threats to Validity The five apps in our study are specific to Python-based web applications using Django, which may not represent all web applications; Other web frameworks, like Rails (Ruby) and Hibernate (Java), let developers specify and use database constraints with similar primitives.

Observation 1: *Many constraints were added as afterthoughts, with 10-72 constraints missed first and added in later pull requests for each application (Table 2).* Such an overlook makes the studied applications vulnerable to invalid data, as it can potentially be stored in the database before the constraints are enforced correctly.

Observation 2: *A majority (82%) of these missing constraints were noticed and added by developers after data integrity issues were detected (Table 3). These issues could lead to severe consequences. Moreover, they took a long time (on average 19 months) to get exposed.* We classify how developers find such missing constraints into four categories: (1) Developers were notified from 30 issue tickets for 31 (22%) missing constraints. Users were likely to have experienced some real-world issues with consequences. (2) After fixing the reported issues, developers sometimes realized that more data fields had similar issues. Thus, they added 59 (41%) such missing constraints. (3) The other 27 missing constraints belong to "Fixed by dev", meaning that developers mentioned "fix", "prevent issue", etc. in comments, which indicates their purpose to fix an issue. (4) 22 (15%) were added due to new features or code refactoring.

We find that 30 different issues with detailed user reports have led to various severe consequences. Among them, 18 issues caused crashes, with 7 of them blocking critical business logic (order or payment-related for e-commerce), causing poor user experience and revenue loss. The other 8 caused data corruption, including order data and other users' account data.

Table 3: Reasons why developers add the *missing* constraints. The majority (82%) originates from issues, either from user reports or developers' findings.

Type	**Related to issue**			**Feature / Refactor**	**Unk-nown**
	From reported issue	Learn from similar issue	Fixed by dev		
Unique	17	16	15	8	3
Not-null	11	40	12	12	1
FK	3	3	0	2	0
Total	31 (22%)	59 (41%)	27 (19%)	22 (15%)	4 (3%)
	117 (82%)				

To make things worse, these missing constraints took a long time (on average 19 months) to be noticed and fixed, opening up a long vulnerable time window that allowed constraint-violating data to be inserted into the database.

Observation 3: *Most (87%) issues that missed database constraints also missed some required checks in the application code. For the rest (13%), even with code checks, the constraint-violating data was still stored during concurrent requests.* It indicates that the code checks are incomplete and insufficient. The 30 issues belong to three categories. (1) 22 (73%) have no checks at all in the application code. (2) Four (13%) issues have checks in some code paths but miss checks in other paths that manipulate the same data. It indicates that developers usually fail to ensure multiple places adhere to the same constraints. (3) Interestingly, for the rest four (13%) issues that have full code checks, constraint-violating data still makes its way into the database. Developers suspected the reason was that code checks failed to handle concurrent requests [72]. They commented, *"This is clearly the result of a race, since we have this check in the view code"*, after careful diagnosis.

Implication In summary, even for these widely deployed web applications, database constraints are not fully leveraged by developers to protect their application data. A large number of database constraints are missing, causing issues with severe consequences. Moreover, the validations in the application code are ad-hoc and generally error-prone to concurrent requests, which makes the situation even worse.

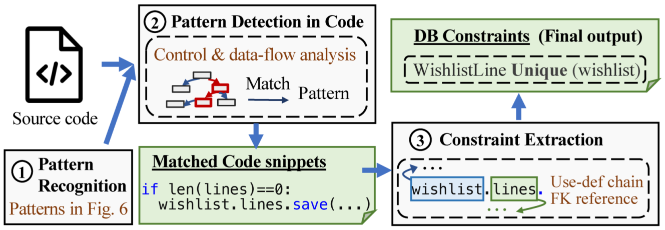
3 DESIGN AND IMPLEMENTATION

3.1 Design Choices: Possible Ways to Find Missing Constraints

Given the consequences brought by missing database constraints, the current practice of adding them after issues have been exposed is far from satisfaction. There are three possible approaches to identify the missing constraints:

Manual inspection Letting developers inspect the whole database schema manually requires their expertise in both database and business logic. It is tedious and error-prone even for domain experts, considering the large number of tables (up to thousands) and columns (up to hundreds per table).

Infer from production data Another approach is to discover from the production data. For example, if a column has a predominant



```
if len(lines)==0:
  wishlist.lines.save(...)
```

Figure 5: The overview of CFinder. CFinder contains three steps to infer the missing database constraints from the application source code. The green boxes are the output of the steps.

percentage of records that satisfy a certain constraint (e.g., 99.99% are not-null), a potential constraint is indicated.

Though the idea is intuitive, it has three main limitations. (1) Approaches based on statistics are usually biased and limited by insufficient datasets. E.g., some rare cases may allow the insertions of null data, but the cases have not been triggered yet. As a result, the wrong conclusion of a not-null constraint can be drawn from the data. Similarly, due to a lack of data, the idea does not apply to newly created tables or added columns, from which most of the missing constraints originate. (2) It is cumbersome for developers to gain access to the production data, especially with access control and privacy concerns. (3) This approach has unacceptably high false positive rates [1, 2]. In previous work [5], 95% of discovered statistically-valid unique constraints are false positives (see more details in §5).

Infer from application code Inferring constraints from the code logic has several advantages. Compared to data, the source code (1) is not limited by data, and (2) contains the business logic of what constraints the data should follow in semantics. Moreover, developers can always cross-check the inferred constraints with the production data. The major concern is, given the code complexity and diversity, *how many data constraints can possibly be inferred from the source code?*

After looking into several real-world web applications, to our surprise, we observe many code patterns that have implicit assumptions on database constraints for all three constraint types. Developers have the assumptions about data constraints in mind, thus their code implementation that retrieves or manipulates the data will follow certain patterns. We list the observed patterns for each constraint type in Figure 6 (§3.3).

Based on the above trade-offs and observations, we choose to extract missing database constraints from the application code.

3.2 CFinder Overview

Figure 5 illustrates the three steps of our approach. In step ① , we recognize the code patterns that imply certain DB constraint assumptions (§3.3). In step ② , with observed patterns and application code as input, CFinder applies control and data flow analysis to find code snippets that match each pattern's conditions (§3.4); In step ③ , from the found snippets, CFinder extracts and infers the formal DB constraints (§3.5); The output of CFinder is the set of missing DB constraints.

The static analysis is flow-sensitive. It is also field-sensitive because CFinder treats the fields of a model class differently. Currently, it does not consider alias. In our evaluation, we didn't catch any false positives caused by aliasing.

3.3 Code Patterns with Assumptions on DB Constraints

3.3.1 Code Patterns. From many web applications, we have observed that many code patterns with assumptions on each constraint type widely exist, but have not been studied before. Figure 6 lists the patterns we discovered, along with real-world examples from e-commerce apps. We name each pattern as $\text{PA}_{\text{(type)(idx)}}$, where type stands for constraint types and idx is the index.

Check existence before save/error handling (PA_{u1} *unique*): The code explicitly checks if the data constraints hold. As Figure 6a shows, it first retrieves records filtered by `product`, then only saves a new record if no existing record returns. It reflects developers' intention on uniqueness: only one record with the value of `product` can exist in DB. Similarly, the pattern can be extended to do error-handling after the check, i.e., throwing exceptions when the record already exists.

APIs with assumptions (PA_{u2} *unique*): Web frameworks provide developers with QuerySet APIs [13] to encapsulate data manipulations. Some APIs are implemented with similar assumptions as *Check existence before error handling*. For example, `get` uses column(s) as the unique identifier to retrieve the record and throws an exception when multiple records are returned [11]. Thus, when developers use this API, they expect the column(s) to be unique. Such APIs include {`get`,`get_or_create`, `get_obj_or_404`} in Django.

Method/field invocation on column without NULL check (PA_{n1} *not-null*): When invoking a method or accessing a field on a column, the column should be not-null. Otherwise, invocation on NULL will throw an exception. We further exclude cases that have explicit NULL checks before the invocation, as the check avoids the exception, making them false positives.

Check NULL before assignment/error-handling (PA_{n2} *not-null*): Similar assumptions as PA_{u1} can be applied to *not-null* with some tweaking. For example, when *order.creator* is null, the app raises an error "Anonymous orders not allowed". One variant is, when the field is NULL, the code explicitly assigns a value to the field before saving, making it not-null.

Field with default value (PA_{n3} *not-null*): Some fields have a default value, which works similarly as PA_{n2}, i.e., assigns the default value to the field if it has not been set before saving. If nowhere in the code would explicitly assign the field a null value, we assume it is not-null.

Column referring primary key (PA_{f1},PA_{f2} *foreign key*): Patterns in Figure 6c reflect the *referential* assumption between tables: the column in the dependent table refers to the primary key (PK) value in the referenced table (PA_{f1}), or vice versa (PA_{f2}). For example, in PA_{f1}, the value from `Voucher` (referenced table)'s PK is saved to `Discount` (dependant table)'s column named `voucher_id`, indicating that `Discount.voucher_id` should be a foreign key to `Voucher`.

Pattern with Assumptions	Control flow graph	Real-world Code Example	Explanation on assumption

PA$_{u1}$ for Unique:
Check existence before save/error-handling
	`if Table(col=val).exist():`
	F
	`Table.save()`
		`/* Oscar: wishlists/models.py */`
		`lines = wishlist.lines.`
		`    filter(product=product)`
		`if len(lines) == 0:`
		`    wishlist.lines.save(...)`
		Implies: WishlistLine **Unique** (product,wishlist)
			Only save record when no record filtered by the column(s) exist.

	`if Table(col=val).exist():`
	T
	`raise Exception`
		`/* Oscar: wishlists/views.py */`
		`if to_wishlist.lines.`
		`  filter(product=product).count() > 0:`
		`     raise Error("WishList already`
		`                containing product")`
		Implies: WishlistLine **Unique** (product,wishlist)
			If one record filtered by the column(s) already exist in database, throws an exception.

PA$_{u2}$ for Unique:
APIs implemented with assumptions
	`Table.get(col=val)` API 'get' implemented as:
	`if len(matched_record)>1:`
	`raise Exception`
		`/* Oscar: dashboard/orders/views.py*/`
		`order = Order.objects.get(`
		`    number=request.GET['order_number'])`
		Implies: Order **Unique** (number)
			Use column(s) as the unique identifier to retrieve data:
			if more than one results exist, throws an exception.

(a) Patterns with assumptions on *Unique* Constraint.

PA$_{n1}$ for Not-null:
Method/field invocation without NULL check
	`Table.col.method()` √
	`if Table.col is not None:`
	`Table.col.method()` X
		`/* Saleor: mutations/draft_orders.py*/`
		`for line in order.lines.all():`
		`  if line.variant.track_inventory`
		`  or line.variant.is_preorder_active():...`
		Implies: OrderLine **Not NULL** (variant)
			Invocation on a column is only valid when the column is *not-null*. Having NULL check before makes it a false positive.

PA$_{n2}$ for Not-null:
Check NULL before assignment /error-handling
	`if Table.col is None:`
	T
	`raise Exception`
	`Table.col=...`
		`/* Shuup: models/_orders.py*/`
		`class Order(Model):`
		`  if not self.creator:`
		`    raise Error("Anonymous orders not`
		`allowed.")`
		Implies: Order **Not NULL** (creator)
			If the column value is NULL, throws an exception or explicitly assign it a value.

PA$_{n3}$ for Not-null:
Field with default value
	`Table.col.default = ...`
	& `Table.col = None` not exist
		`/* Oscar: order/models.py */`
		`class OrderLine(Model):`
		`  quantity = IntegerField(default=1)`
		Implies: OrderLine **Not NULL** (quantity)
			The column has a default value assigned and no place sets the column to be null.

(b) Patterns with assumptions on *Not-null* Constraint.

PA$_{f1}$ for FK:
Column referring primary key
	`DepTable(col=RefTable.pk).save()`
	`DepTable.filter(col=RefTable.pk)`
		`/* Oscar: apps/order/utils.py */`
		`def create_discount_model(self):`
		`  order_discount.voucher_id = voucher.id`
		`  order_discount.save()`
		Implies: Discount **FK** (voucher_id) **ref** Voucher(id)
			Dependent table's column is assigned the value of Referenced table's primary key.

PA$_{f2}$ for FK:
Primary key referring column
	`RefTable.get(pk=DepTable.col)`	`/* Saleor: mutations/products.py */`
		`class ProductVariantDelete():`
		`  product = Product.get(`
		`        id=instance.product_id)`
		Implies: Variant **FK** (product_id) **ref** Product(id)
			Use Dependent table's column value when filter by Referenced table's primary key.

(c) Patterns with assumptions on *Foreign Key* Constraint

Figure 6: Code patterns with implicit assumptions on three DB constraint types, together with real-world examples and explanations.

Our evaluation in §4.2 and §4.3 shows that these code patterns are effective for detecting missing DB constraints (found 210 previously unknown constraints) and have good coverage (79% recall on a collected dataset). We also discuss potential improvements and extensions to the patterns there. Besides, since these patterns reflect semantic code logic, they are general and applicable to applications in other frameworks or languages.

3.3.2 Conditions of Code Patterns. After observing these code patterns, a natural question would be how to detect them in the application code. A naïve way is to represent the patterns with some predefined regular expressions and match the code with them. This may work for simple cases with well-defined APIs, such as get. But it cannot detect most other cases. Take PA$_{n2}$ "check NULL before assignment" as an example, matching any assignment after any NULL check would introduce too many false positives, since the two operations could come from unrelated code blocks and operate on unrelated data. Such complex control and data logic can hardly be defined and matched with regular expressions. Moreover, it cannot infer the table of the constraints as that requires the data flow information (§3.5).

Instead, CFinder represents patterns as the conjunction of three types of conditions, which involve control and data dependencies built on top of the abstract syntax tree (AST) [49]. Based on it, our detection algorithm (§3.4) traverses the AST and finds snippets that match all conditions of a pattern.

To introduce the three types of conditions, we use the first pattern PA_{u1} for unique constraint in Figure 6 as the example, which *checks existence before save/error handling*.

Control dependencies (C-D) Each pattern consists of several sub-components (subtrees in AST), and each subtree has its specific semantic meaning. These subtrees follow certain control dependencies. For example, PA_{u1} requires two sub-components, *check existence* and *save*. They represent two subtrees that satisfy the control flow of the *IF* block, i.e., `condition` for *check existence*, and `body` or `else` for *save*.

Other types of control dependencies include one syntax tree T_i being the parent of another tree T_j, etc. Using another pattern PA_{n1} as the example, we require that for all parent trees of the *field invocation*, no one T has a condition branch T_{cond} that has the NULL check.

Syntax pattern matching (P-M) As we mentioned, each subtree needs to represent a specific semantic meaning. To bridge their gap, we *pre-define* a set of syntax-based patterns P_*, where each P_* consists of a category of simple syntax tree patterns with the same semantic meaning S_*. Therefore, whether a subtree T_* represents a semantic meaning S_* can be evaluated by T_* matching with one syntax pattern of P_*.

For example, we define P_{exist} to represent the category of patterns indicating a check on the `existence` of a record. One such syntax tree could be a `Call` block with a `Attribute` subtree with name `exist` (Check more in Figure 7). These syntax patterns are general to the framework and easy to customize.

Back to PA_{u1}, it requires: (1) the if-condition checks whether the record *exist* or *not exist*, i.e., T_{cond} matches with P_{exist} or $\neg P_{\text{exist}}$. (2) Respectively, the two subtrees T_{body} and T_{else} in two branches match with P_{save} (save record when not exist) or P_{error} (error-handling when a record exists). The results (R_*) of these syntax pattern matching are connected with AND and OR, to form the final evaluation of this condition.

Data dependencies (D-D) This condition requires the data in subtrees to follow certain data dependencies, i.e., the two subtrees operate on the same tables and columns.

In PA_{u1}, we require the match of the table and column that (1) get saved in T_{body} and (2) perform the NULL check in T_{cond}. We evaluate the data dependencies by first inferring those tables and columns from each subtree using data-flow analysis (§3.5) and then matching them.

To sum up, we list the formal representation of PA_{u1}:

$(C-D)$ $[T_{\text{cond}}, T_{\text{body}}, T_{\text{else}}] = \text{IF\_block\_subtrees}()$

$(P-M)$ $R_{\text{cond}} \land (R_{\text{body}} \lor R_{\text{else}})$, where

$$[R_{\text{cond}}, R_{\text{body}}, R_{\text{else}}] =$$
$$\text{MATCH}([T_{\text{cond}}, T_{\text{body}}, T_{\text{else}}], [P_{\text{exist}}, P_{\text{error}}, P_{\text{save}}]) \lor$$
$$\text{MATCH}([T_{\text{cond}}, T_{\text{body}}, T_{\text{else}}], [\neg P_{\text{exist}}, P_{\text{save}}, P_{\text{error}}])$$

$(D-D)$ $\text{DataDepend}(T_{\text{cond}},\ T_{\text{body}} \lor T_{\text{else}})$

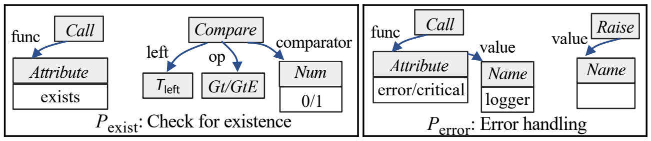


Figure 7: Example of pre-defined syntax tree patterns. We use them to match with the candidate syntax trees. Each category of P_* can have several patterns representing the same semantic meaning.

3.4 Code Patterns Detection Algorithm

In step ② , CFinder detects code snippets that can match the conditions of one code pattern from the application code. Taking the first code snippet for PA_{u1} in Figure 6a as the example, we show how it can be detected from the code.

3.4.1 *Overall Algorithm.*

The steps are as follows.

- CFinder walks the module’s AST in a breadth-first fashion to identify the candidate code snippets whose root types match the pattern’s root type (`IF` node in PA_{u1}).
- For each code snippet, CFinder then extracts their subtrees following the *control dependency* of the pattern, i.e., extracts subtrees T_{cond}, T_{body},T_{else} from the root `IF` node.
- CFinder then performs the *syntax pattern matching* on each subtree. E.g., match subtree T_{body} (`wishlist.lines .save`) with predefined P_{save} (details in next paragraph).
- CFinder further checks the *data dependencies* using the use-definition graph to see if variables in two subtrees refer to the same table and columns (details in §3.5).
- If all pattern conditions evaluate to True, then we find a candidate snippet with assumptions on DB constraint.

3.4.2 *Match Subtree with Syntax Pattern.*

Figure 8 shows the syntax tree of the example snippet on the left, with some subtrees collapsed. The `MATCH` function matches its subtree T_{body} (left) with the predefined syntax pattern P_{save} (right).

Here, P_{save} represents the category of syntax patterns that have the meaning of “saving a record”. In the AST form, one example of the syntax pattern is a `Call` node calling an `Attribute` node named `save` or `create`.

To implement `MATCH`, CFinder performs a breath-first traversal in T_{body} and finds the node which matches the root of P_{save}, i.e., the `Call` node. Then for each child node of `Call` in P_{save} (the `Attribute` node), CFinder checks if there is a corresponding subtree node in T_{body}. CFinder recursively repeats this process until the leaf nodes of P. If all children have a match, CFinder concludes that T_{body} matches P_{save}.

Figure 7 shows more examples: two categories of pre-defined syntax patterns for P_{exist} and P_{error}. Note that these patterns are as simple as a syntax tree with a depth of only one or two, and they have no control or data dependencies. We collect them heuristically by studying the application code. They are general to applications, and *more importantly*, they can be easily customized and extended.

3.5 Database Constraints Extraction

In step ③ , CFinder automatically converts the snippets into formal DB constraints. After detecting the second code snippet in

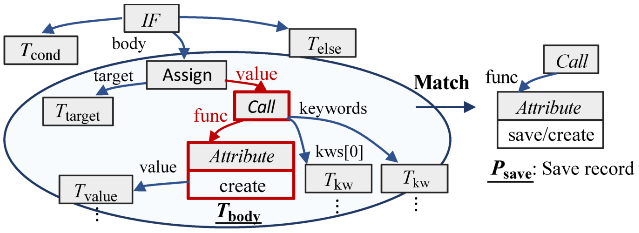


Figure 8: Matching syntax tree T (left) with pre-defined syntax pattern P (right). The subtree with bold red borders on the left is the match. T (left) is constructed from the example in Figure 6a.

Figure 6a that matches PA_{u1}, in Figure 9, this step infers the table WishListLine and columns (wishlist,product) from it. To achieve it, CFinder traces the definitions in code using use-definition analysis [29] and table metadata.

3.5.1 Identify the Table. The MATCH step identifies the variable list (to_wishlist.lines) that represents the table object. However, identifying which table it represents (WishListLine) is often *non-trivial* due to two challenges:

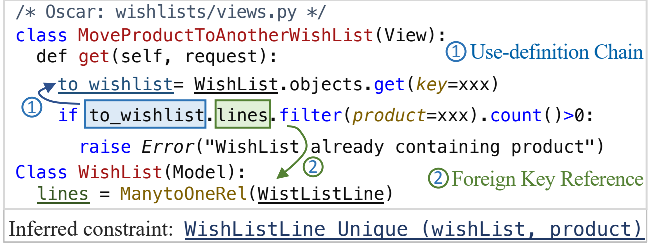
- Python's language feature, dynamic typing, i.e., a variable's type is defined at runtime. Therefore, static analysis doesn't know to_wishlist's type (class of WishList table).
- The "variable" may involve a chain of field accesses, which transfers from one table to another following the foreign key reference. For a real example in Oscar, self.attribute. option _group. options involves the reference between three tables. It is hard to sort out the relationship with such complex code even with human inspection.

To handle the first challenge, CFinder *infers the definition using use-def chain analysis*. Starting from the first variable, CFinder traces its definitions in the use-definition chain and identifies one of the definitions being table class. In the example, to_wishlist gets the definition from WishList .objects.get, which returns an instance of WishList class. To be scalable to large applications, CFinder does not perform the inter-procedure analysis.

Second, CFinder follows the list of field accesses and tracks the corresponding tables using the table's metadata. Starting from to_wishlist, which is an instance of WishList class, .lines retrieves the instance of a WishListLine class through the foreign key reference. CFinder repeats this process until the end of the field list.

3.5.2 Identify the Column. The columns of the not-null and foreign key constraints are usually obvious and CFinder gets them directly from the specified patterns. Here we discuss two special cases, i.e., *composite* and *conditional* unique constraints:

- When retrieving referenced objects through the foreign key field, it contains the *implicit join* on table ID. In the example, to_wishlist.lines retrieves the lines related to the to_wishlist instance. Consequently, besides product, the generated SQL statement filters on wishlist_id as well. Thus, CFinder infers that the final constraint requires columns (wishlist, product) to be *composite unique*.



```
/* Oscar: wishlists/views.py */
class MoveProductToAnotherWishList(View):
  def get(self, request):
    to_wishlist= WishList.objects.get(key=xxx)
    if to_wishlist.lines.filter(product=xxx).count()>0:
      raise Error("WishList already containing product")
Class WishList(Model):
  lines = ManytoOneRel(WistListLine)
```

Inferred constraint: WishListLine Unique (wishList, product)

Figure 9: Infer the constraint table from code: (1) Infer the definition using data flow analysis (2) Track the tables along the chain of field accesses. Specifically, to_wishlist represents the WishList class and to_wishlist.lines refers to the final WishListLine class through the foreign key.

- When retrieving records by filtering on columns with fixed values (e.g., filter(col,valid=True)), it indicates a "partial (conditional) unique constraint" [47], which restricts the uniqueness of col over a subset of data defined by the condition (valid=True).

3.5.3 Get Missing DB Constraints. After inferring all DB constraints from the code, CFinder filters the existing constraints retrieved from information_schema tables of databases.

4 EVALUATION

As shown in Table 4, we evaluate CFinder on eight large web applications including seven widely-adopted open source web applications and the main web application of one commercial enterprise (Company) with millions of end-users. The open-source applications are top-starred in each category on Github, with three of them having 10K stars and five having 5K stars. Moreover, Saleor is adopted by e-commerce companies including one with 50M revenue [57], Edx by 160 institutes and has millions of users [21], Zulip by large communities and universities [75], etc. These open-source applications have 74K to 617K LOC, more than 60,000 commits, and have high demands on data integrity and reliability due to their wide adoption and millions of users. We use the latest version of all applications (commit hashes are in the references).

We evaluate the effectiveness of CFinder based on how many new *missing* database constraints can be detected (§4.1). We further report them to the app developers and get their confirmations (Table 4).

Moreover, we evaluate the precision of the detected missing constraints (§4.2) and study the reasons for false positives. We have two human inspectors independently examine the detected missing constraints and label a case as true positive only when consensus was reached. Furthermore, we evaluate the coverage (recall) of CFinder (§4.3) on two datasets. The first dataset contains all the existing (not missing) database constraints already set by the latest application code. The second dataset contains 117 real-world *missing* constraints collected from the past commit history (Table 3). These missing constraints were noticed because data integrity issues were detected. We further evaluate CFinder's performance (§4.4) and discuss the developer's feedback (§4.5).

Table 4: Evaluated applications and detected missing DB constraints from them. *"Detected existing"*: Detected constraints that already exist in DB. *"Detected missing"*: Detected constraints that miss in DB. *"ACK by dev"*: numbers of missing constraints acknowledged by developers that need to be added. For three apps with zero confirms, we received no response to our issue reports.

App.	Category	Github stars	LOC	Detected existing	Detected missing	ACK by dev
Oscar [44]	E-comm	5.2K	74K	159	**24**	**5**
Saleor [56]	E-comm	15.3K	298K	220	**15**	**0**
Shuup [58]	E-comm	1.8K	196K	290	**31**	**0**
Zulip [79]	Team chat	15.3K	361K	265	**21**	**12**
Wagtail [63]	CMS	11.7K	181K	69	**10**	**7**
Edx [42]	Online course	6K	617K	509	**43**	**0**
EdxComm [22]	E-comm	122	93K	97	**14**	**6**
Company	Enterprise	-	-	-	**52**	**45**
Total	-	-	-	1609	**210**	**75**

Databases are fully set up and populated with testing data only. All experiments are done on a single machine with a 2.30GHz CPU (6 core), 16GB Memory and 256GB SSD running a Ubuntu 18.04 distribution.

4.1 Effectiveness in Detecting *Missing* DB Constraints

4.1.1 Overall Results. Table 4 shows the number of detected missing database constraints from each web application. Overall, CFinder detects 210 missing database constraints from eight web applications, including 10-43 missing constraints for each open-source web application and 52 missing constraints for a commercial company with millions of users.

We manually validated the detected constraints and reported the identified true missing constraints to app developers. When we contacted the developers, we prioritized these applications that actively responded to our issue reports. For three apps with zero confirms, we received no response to our reports.

So far we reported 92 of them and we have got **75** confirmed by developers as real missing database constraints, including 30 of them from seven open-source web applications and 45 from the commercial company. Among the 75 confirmed constraints, there are 37 unique constraints, 22 not-null constraints, and 16 foreign key constraints. We provided one example for each constraint type in Table 5 to demonstrate the potential consequence of not having the missing constraints.

4.1.2 Breakdown of the Detected Missing Constraints. To understand the effectiveness of CFinder in detecting each type of *missing* database constraints, we present the breakdown for different code patterns in three constraint types, *Unique*, *Not-null*, and *Foreign key* in Table 6.

- *Unique constraint:* CFinder detects 66 missing *unique* constraints, with two code patterns detecting 16 and 56 respectively. Moreover, among them, 13 are "partial unique constraints" (§3.5). Some app developers are not aware of this type of constraint, thus not taking advantage of them.
- *Not-null constraint:* For total 77 detected constraints, three patterns detect 44, 11, 22, respectively.
- *Foreign key constraint:* CFinder detects 15 missing *foreign key* constraints in total. The number is relatively small, which is consistent with our study (§2) on real-world missing constraints in history. A possible reason is that when developers use the field to reference another table, the referential relationships are usually so obvious that developers are unlikely to neglect them.

Table 5: Examples of confirmed missing database constraints. The first two examples are already merged in their main branches.

Confirmed Unique constraints (37 cases)	
Example:	`ProductAttr Unique(code,product_class)` [19]
Potential consequence:	Product attributes with same attribute code for a product class are invalid and invisible to customers.
Confirmed Not null constraints (22 cases)	
Example:	`Attachment Not NULL (realm)` [76]
Potential consequence:	The attachment is not valid when uploaded without a realm (organization). Similar as a data loss to users.
Confirmed Foreign Key constraints (16 cases)	
Example:	`OrderDiscount (offer) Ref Offer (id)` [20]
Potential consequence:	The discount on an order is not valid without linking to an existing offer.

Table 6: The breakdowns of the number of detected *missing* database constraints for each constraint type and code pattern. One constraint can be detected by multiple code patterns, and we only count them once in *Tot.*(Total).

App.	Detected *missing* constraints									
	Unique			Not null				Foreign Key		
	PA_{u1}	PA_{u2}	**Tot.**	PA_{n1}	PA_{n2}	PA_{n3}	**Tot.**	PA_{f1}	PA_{f2}	**Tot.**
Oscar	3	10	12	9	1	0	10	1	1	2
Saleor	2	3	5	7	0	1	8	1	1	2
Shuup	2	4	6	12	5	7	24	1	0	1
Zulip	5	7	10	2	1	4	7	2	2	4
Wagtail	0	4	4	2	0	4	6	0	0	0
Edx	3	22	23	6	3	6	15	1	4	5
EdxComm	1	6	6	6	1	0	7	0	1	1
Total	16	56	**66**	44	11	22	**77**	6	9	**15**

4.2 False Positives in Detected Missing DB Constraints

As Table 7 shows, CFinder's precision in detected missing constraints is reasonably high for all three types of database constraints, 82%, 75%, 80% for unique, not-null, and foreign key constraints, respectively.

In total, 34 false positives (FPs) are introduced. There are two main reasons. *First*, 12 (35%) FPs are caused by the static analysis being unsound. Five have wrongly inferred database tables (§3.5) and seven have unrecognized or implicit NULL checks before the field invocation (thus these columns could be NULL without throwing exceptions). These FPs could be mitigated by fine-tuned code analysis, such as incorporating the inter-procedure information. *Second*, 13 (38%) FPs are caused when code matches the pattern but contains no assumption on constraints. For example, one code

Table 7: The precision of detected missing constraints by CFinder. *Tot.*: Total number of detected missing DB constraints. *Precision*: $\frac{\text{TruePositive}}{\text{TruePositive+FalsePositive}}$.

App.	Unique			Not null			Foreign Key		
	Tot.	TP	**Precision**	Tot.	TP	**Precision**	Tot.	TP	**Precision**
Oscar	12	9	75%	10	8	80%	2	2	100%
Saleor	5	3	60%	8	7	88%	2	2	100%
Shuup	6	5	83%	24	17	71%	1	1	100%
Zulip	10	7	70%	7	5	71%	4	2	50%
Wagtail	4	4	100%	6	4	67%	0	0	-
Edx	23	20	87%	15	11	73%	5	4	80%
EdxComm	6	6	100%	7	6	86%	1	1	100%
Overall	66	54	**82%**	77	58	**75%**	15	12	**80%**

snippet satisfies the pattern PA_{u1}, but it is only meant for a sanity check to handle a special case of no valid voucher, which does not involve uniqueness assumptions. To prune those FPs, CFinder can further refine the patterns with finer-grind semantics.

4.2.1 Impacts of False Positives. The reported FPs can be easily recognized by developers, thus will not cause serious consequences. (1) Developers won't be misled after checking the code snippet (reported by CFinder) that implies the constraint. For example, developers who read the error message can easily determine if it warns about a constraint violation. (2) Developers can run simple scripts to automatically check if the constraint is consistent with the production data, i.e., using data-driven approaches as complementary. (3) Even if they wrongly add a constraint, the DBMS will reject the schema migration if any existing data violates it. Developers then decide whether this is a FP or if data cleaning is required. In either case, if a constraint can be added, existing data must adhere to the constraint already.

4.2.2 Human Inspection Efforts. (1) It took two graduate students about 40 hours to manually inspect the FPs from 158 constraints that CFinder reported in open source applications. Most time is spent on understanding how the field is used all over the codebase. (2) Based on our interactions with app developers, they are familiar with code and they do thorough inspections including the production data. The inspection time is acceptable. Half of the missing constraints we reported to Zulip's work channel are diagnosed within 20 minutes.

4.3 Coverage of Database Constraints

We then evaluate the percentage of database constraints that CFinder can cover in its detection, i.e., the recall of CFinder, on two different datasets. We further look into what are missed by CFinder.

4.3.1 Evaluation with Existing DB Constraints. Even though the goal of CFinder is to detect the *missing* constraints, we can evaluate whether the *existing* constraints behave consistently with the code patterns. Specifically, we evaluate how many *existing* DB constraints already set in the database can be covered by CFinder. It reflects the generalization of the patterns. Note that we exclude foreign keys, as the existing ones are used differently from the patterns for missing ones. Specifically, for foreign keys that already exist in DB, developers mostly retrieve the referenced table through field invocations, such as `order.product` when `product` is a FK.

Table 8: The percentage of *existing* constraints already set in the database that CFinder can cover in the detection.

App.	# *Existing* constraints already set		**What percentage can CFinder cover**	
	Unique	Not null	Unique	Not null
Oscar	49	156	67%	81%
Saleor	70	210	74%	80%
Shuup	89	298	70%	77%
Zulip	47	278	72%	83%
Wagtail	18	79	61%	73%
Edx	133	569	65%	74%
EdxComm	30	110	67%	70%

Table 9: The percentage of *missing* constraints that CFinder can cover in the collected dataset. The dataset from our study (§2) contains constraints that missed first and added by later commits.

# *Missing* constraints in dataset			**What percentage can CFinder cover**		
Unique	Not null	Foreign Key	Unique	Not null	Foreign Key
48	63	6	79%	83%	50%

Table 8 shows that CFinder has a reasonable recall. It can detect 61%-74% of unique constraints and 70%-83% of not-null constraints for seven web applications.

We randomly sample and study 40 false negatives for each of the two constraint types. They belong to three categories: (1) 57 (71%) do not exhibit any general patterns with assumptions on constraints. Among them, 20 are fields used for specific purposes and they might be improved by incorporating some application-specific domain knowledge. For example, some fields are used in the URL as the identifier, which may imply uniqueness. (2) 17 (21%) are fields not used in the application code logic, just placeholders for legacy or future use. (3) 6 (8%) have usages with assumptions but are not detected, mainly because code patterns are hard to recognize when they span different functions in the call chain. These can be improved by tracing the inter-procedure information.

4.3.2 Evaluation with Dataset on Missing Constraints. In our study (§2), we collect a dataset of 117 *missing* database constraints from the schema migration history (Table 3). These *missing* constraints were noticed after having some data integrity issues that caused real damage. We evaluate if CFinder can detect these missing constraints on old versions of code, which could help *prevent the issues from happening*.

Table 9 shows that CFinder has a good coverage. Out of the 117 real-world missing constraints in the dataset, CFinder can detect 93 (79.5%) of them. These missing constraints would be caught if CFinder had been adopted. We failed to detect 24 constraints mainly because they are too specific, i.e., do not exhibit general patterns. Note that we also mark those constraints that "learned from similar issues" as detected if the original issue is detected.

4.4 Performance of CFinder

CFinder is designed to run in the testing environment thus its performance is not time-critical. Table 10 shows that the analysis

Table 10: Time (seconds) to run CFinder's static analysis.

App.	Oscar	Saleor	Shuup	Zulip	Wagtail	Edx	EdxComm
Analysis time (s)	22	64	75	59	40	147	30

time of CFinder's static code analysis is less than 150 seconds for each application, and is near proportional to the application's lines of code (up to 620K LOC for Edx).

4.5 Developers' Feedback Discussion

We reported 92 of the detected constraints to the application developers and have got 75 confirmed so far. The others are rejected or still under investigation. Here we share the experience of the interactions with developers.

We are encouraged by the positive feedback from many developers of the evaluated applications. For example, Zulip developers quickly responded to our reported issues and actively examined their code base for similar issues with us [76, 78]. The confirmed missing constraints were either due to a lack of considerations in the design, or due to missing checks after business requirements changes. As one developer replied in the report for a not-null constraint, *"Being after that migration has run, ...,there's no reason to keep it nullable"*.

In contrast, we find that maintainers hesitated to enforce some missing constraints we reported. For example, in one issue [45], the developers worried that the data migration might take too long a time to process the null values for large tables. In another issue, the developers assume that the invalid record will not be generated during normal workloads in current code logic, and thus are reluctant to add fixes [77].

5 RELATED WORK

Empirical study of data constraints in web applications Previous studies have investigated the adoption of data constraints in the *application layer* [4, 69]. Bailis et al. [4] study the effectiveness of *application-level validations* as substitutes for their respective database constraints counterparts in web frameworks (Rails). Their quantitative experiment shows that app validations lead to data corruption due to concurrency errors in 13% of usages. Yang et al. [69] study the location, expression, and evolution of data constraints. They find that developers struggle with maintaining *consistent* data constraints among the front-end browser, the application (using framework's validation APIs), and the database. In contrast, our study in §2 focuses on the *missing* constraints neglected by developers in the database layer, which motivates tooling support to systematically detect the missing constraints.

Detecting data dependencies from applications Yang et al. [69] study the constraints specified in framework's validation APIs and their inconsistencies with constraints in the database. Liu et al. [35] detect constraints specified in framework's validation APIs in model classes with the motivation to use constraints to optimize query execution performance.

Our work differs largely in the following ways. (1) These works require developers to *already know and specify* these constraints using validations. In other words, they cannot help with the *missing database constraints neglected by developers*. Thus, our identified missing constraints cannot be discovered by their works. Besides, in order to infer from the code logic with implications, CFinder proposes more advanced code analysis algorithms. (2) Their goal is to optimize the performance or study inconsistencies, while CFinder proposes to enforce the *missing* DB constraints to protect the data integrity. (3) Majority (88%) of their detected constraints are defined only in the framework level and are not DB built-in constraints, as they stated "defining inclusion and format constraints requires writing UDFs, which is tedious to implement in most DBMS" [35]. Thus they are orthogonal to CFinder.

Inferring constraints from data Previous works on data profiling [1, 2] discover the data constraints by collecting statistics about the data itself. Aside from the limitation of biased and insufficient datasets we discussed in §3.1, these works still lack effective techniques to discover missing constraints that apps truly require. Specifically, as unique or foreign key constraints involve multiple columns, they traverse the search space of a powerset of column combinations and validate if the data satisfies the constraint. A majority of works focus on pruning the search space [3, 5, 30, 46, 70]. However, it is understudied which of the discovered *statistically-valid* constraints are truly required by apps in semantics. In fact, a vast majority (>95%) of them are false positives [2, 5]. Some [26, 53, 70] propose heuristic rules to prune FPs, but their effectiveness lack evaluations on real-world large datasets.

In contrast, the source code (1) is not limited by data quality, and (2) reflects what constraints the data really needs to follow in semantics. The evaluation shows CFinder introduces reasonable precision (78%) and recall (79%).

Invariant detection from trace The line of work on invariant detection tools, like Daikon [24, 25], dynamically traces program runtime states and infers likely invariants in code. Typically dynamic approaches have a challenge of coverage problem. For likely invariants, the coverage problem of test cases or product runs can also lead to many false positives and false negatives, particularly false positives.

Application verification and synthesis using constraints. Another line of work focuses on using data constraints for program verification and synthesis. Li et al. [33] detect the application bugs that violate the numerical data assertions inferred from the data. Wang et al. formally verify the equivalence of programs with different DB schemas [64] and synthesize equivalent programs [65]. These works are orthogonal and may help with code evolution when adding new constraints.

Leveraging constraints to improve performance and security Various constraints have been used to find better query plans and optimize query performance [34, 35, 67]. Our work reveals that there are opportunities to find more required database constraints, thus could complement their works.

Some works study methods to impose and verify the security and privacy "policies" [32, 41, 68]. These policies are usually too complex to be supported by current databases, thus are orthogonal to our work. Future work can study the automatic detection of these missing privacy-related policies from code. They are promising to improve the data quality in the further.

6 LIMITATION & DISCUSSION

CFinder targets web applications that are backed by RDBMS and have a high requirement on data integrity, which widely exist in our daily life. Some systems shift the responsibility of data quality to the application layer as a design choice for better scalability and customization. It includes apps backed by NoSQL databases, which typically do not support constraints in DB. Though not our targets, CFinder can still benefit them by identifying the missing data constraints and helping them check at the application/framework level. Moreover, NoSQL databases such as MongoDB recently start to support constraints at the database [39] level, showing its importance and potential.

CFinder is currently implemented for Python-based web applications, as it relies on web frameworks' APIs to identify database operations when performing pattern matching in §3.4. For example, we use Django's five APIs for record retrieval, three for record creation or updating, and one for existence check. However, CFinder's code patterns in §3.3 are *general* as they catch the semantic assumptions on data constraints in code logic. We also studied Rails (Ruby), Flask (Python), and Hibernate (Java), and they all encapsulate similar sets of APIs for the four database operations. Thus, CFinder can be migrated to other frameworks or languages with reasonable implementation efforts.

Adding the missing constraints may require extra efforts to clean the data if application data is already erroneous or incompatible. The overhead to perform data cleaning and migration sometimes is not negligible for large tables. However, we consider the effort essential and beneficial because these corrupted data could lead to serious business loss in the future.

Like most issue detection tools, CFinder still has false positives (§4.2) and false negatives (§4.3), and there is still space for further improvement. The false negatives could be improved by extending CFinder with more application-specific code patterns and fine-tuning the static analysis. To avoid the false positives, we would have to rely on developers to manually examine their semantics in code. CFinder can perform more refinement steps in the static analysis to prune those false positives.

7 CONCLUSION

In this paper, we focused on the problem of missing database constraints in web applications with resulting data integrity issues and the feasibility of extracting the missing database constraints from the application code. Specifically, we first conducted an empirical study on *missing* constraints in five popular web applications. Then we designed and implemented a tool that identified 210 previously unknown missing constraints with reasonable accuracy from eight widely-deployed web applications, including one commercial company with millions of users. We have reported 92 of them to the developers of these applications, so far 75 of them are confirmed.

ACKNOWLEDGMENTS

We greatly appreciate the anonymous reviewers for their insightful comments and feedback. We thank Andrew Yoo and Tianwei Sheng from Whova Inc for helping with the evaluation. We thank Chengcheng Xiang, Tianyi Shan, Eric Mugnier from the Opera group at UC San Diego for useful discussions and paper proofreading. This work is supported in part by NSF grants (CNS-1814388, CNS-1526966) and the Qualcomm Chair Endowment.

A ARTIFACT APPENDIX

A.1 Abstract

CFinder is a static analysis tool that analyzes application source code to automatically infer and detect any missing database constraints to improve the database integrity. Its workflow contains three steps:

- With the application code as input, CFinder applies the proposed static analysis to find the code snippets that match the conditions of code patterns with assumptions on database constraints.
- From the found snippets, CFinder extracts and infers the formal DB constraints.
- After comparing them with the existing database schema, CFinder outputs the set of missing database constraints.

CFinder reports the inferred missing database constraints with detailed code pattern information. We provide an artifact, described in detail below, to help the easy reproduction of all the key evaluations in section 4 of the paper. The artifact is available on GitHub at https://github.com/huanghc/cFinder.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** Static code analysis
- **Program:** We release the source code of CFinder in the artifact and evaluate CFinder with seven open-source Python-based web applications.
- **Data set:** Source code and database schema of seven open-source web applications.
- **Run-time environment:** Ubuntu with Python 3.8
- **Metrics:** The number of detected missing and existing database constraints.
- **Output:** The script will output the detected database constraints from the source code, their coverage, and their code pattern information.
- **How much disk space required (approximately)?:** 5 GB disk should be enough for the experiments. This will include the source code of our tool, the source code of seven web applications, all database schema data, and all generated results.
- **How much time is needed to prepare workflow (approximately)?:** It takes about 15 minutes. The whole workflow takes one script to launch. Time will be used to set up the Python runtime environment and download the source code of the evaluated applications.
- **How much time is needed to complete experiments (approximately)?:** It takes about 10 minutes. The workflow takes one script to launch. Time will be used to run the static code analysis.
- **Publicly available?:** Yes [71].
- **Code licenses (if publicly available)?:** MIT
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.7425382

A.3 Description

A.3.1 How to Access. The artifact is available on GitHub: https://github.com/huanghc/cFinder.

A.3.2 Hardware Dependencies. Our tool and the experiments should be run on a Linux machine with at least 8 GB RAM and 4 cores.

A.3.3 Software Dependencies.

- Linux (we tested on Ubuntu 18.04)
- Python >=3.8

A.3.4 Data Sets. The artifact evaluates seven open-source web applications. Our scripts will automatically download their source code from GitHub. The artifact includes (1) The files containing the database constraints and schema of these web applications (in the directory `data/`). These data are used in the static analysis to generate the main results. (2) The similar files containing the database constraints and the source code for the history issues (in the directory `data/history_issues`). These data are used for Table 9 only.

A.4 Installation

We provide a `make install` command to automatically finish the following steps: (1) Pull the application source code from Github; (2) Set up the Python virtual runtime environment.

A.5 Experiment Workflow

We provide a `make run_all` command to automatically perform all the evaluations with the following steps: (1) CFinder applies static code analysis to find the code snippets that match the proposed code patterns. From the code snippets, CFinder extracts and infers the formal database constraints. (2) After comparing them with the existing database schema, CFinder outputs the set of detected missing DB constraints and the set of existing constraints that CFinder can cover. (3) CFinder also runs the same static analysis again on the history issues' dataset.

A.6 Evaluation and Expected Results

We provide the scripts to automate the evaluation and generate the Tables and numbers in §4. The output will be in the `result/` directory and contain the CSV files with the following key results:

- The total number of detected existing and missing database constraints from each application. (Table 4)
- The breakdown of the number of detected missing database constraints for each constraint type. (Table 6)
- The percentage of existing constraints already set in the database that CFinder can cover. (Table 8)
- The percentage of missing constraints in the collected dataset that CFinder can cover. (Table 9)
- Time (seconds) to run the static analysis. (Table 10)

More detailed results for the detected database constraints of each application and each constraint type are in the `result/APP_NAME/` directory:

- `newly_detected.csv` contains all the newly detected constraints with their code pattern information.
- `existing_constraints.csv` contains the existing constraints in the database that CFinder can cover.

Note that some results involve human inspection (Table 7) and developers' confirmation (last column in Table 4), thus not included in the artifact. Note that due to the differences in hardware environments, the performance results in Table 10 can be different from the numbers reported in the paper.

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling relational data: a survey. *The VLDB Journal* 24, 4 (2015), 557–581.

[2] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2016. Data profiling. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 1432–1435.

[3] Ziawasch Abedjan and Felix Naumann. 2011. Advancing the discovery of unique column combinations. In *Proceedings of the 20th ACM international conference on Information and knowledge management*. 1565–1570.

[4] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2015. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1327–1342.

[5] Johann Birnick, Thomas Bläsius, Tobias Friedrich, Felix Naumann, Thorsten Papenbrock, and Martin Schirneck. 2020. Hitting set enumeration with partial information for unique column combination discovery. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2270–2283.

[6] Nedyalko Borisov and Shivnath Babu. 2011. Proactive detection and repair of data corruption: Towards a hassle-free declarative approach with amulet. *Proceedings of the VLDB Endowment* 4, 12 (2011), 1403–1406.

[7] Alex Bunardzic. 2021. Should Database Manage The Meaning? Retrieved Feb 20, 2022 from http://lesscode.org/2005/09/29/should-database-manage-the-meaning/

[8] Ting Dai, Jingzhu He, Xiaohui Gu, Shan Lu, and Peipei Wang. 2018. Dscope: Detecting real-world data corruption hang bugs in cloud server systems. In *Proceedings of the ACM Symposium on Cloud Computing*. 313–325.

[9] Christopher John Date. 1975. *An introduction to database systems*. Pearson Education India.

[10] Django. 2021. Constraints reference. Retrieved Nov 12, 2021 from https://docs.djangoproject.com/en/4.0/ref/models/constraints/

[11] Django. 2021. get() in django/db/models/query.py. Retrieved Nov 12, 2021 from https://github.com/django/django/blob/d8b437b1fbe3bf54822833bea5e19d2142cf3e1f/django/db/models/query.py#L499

[12] Django. 2021. The Web framework for perfectionists with deadlines | Django. https://www.djangoproject.com/

[13] Django. Jan. 2022. QuerySet API reference. https://docs.djangoproject.com/en/4.0/ref/models/querysets/.

[14] Django. Nov. 2021. QuerySet API reference - get_or_create(). https://docs.djangoproject.com/en/4.0/ref/models/querysets/#get-or-create.

[15] Django. Nov. 2021. Writing database migrations. https://docs.djangoproject.com/en/3.2/howto/writing-migrations/.

[16] Django-oscar. 2021. Change type and name of basket field on AbstractOrder. Retrieved Nov 12, 2021 from https://github.com/django-oscar/django-oscar/commit/9fa2589b6c70d1f3bff381233eddc41a63aa22e4

[17] Django-oscar. 2021. Check for existing email when updating profile. Retrieved Nov 12, 2021 from https://github.com/django-oscar/django-oscar/pull/324

[18] Django-oscar. 2021. Why is order.basket_id not a ForeignKey? Retrieved Nov 12, 2021 from https://groups.google.com/g/django-oscar/c/M0FgIB_f9tM/m/W-52L1zZMxAJ

[19] Django-oscar. 2022. Make attribute codes unique per product class. Retrieved Mar. 20, 2022 from https://github.com/django-oscar/django-oscar/pull/3823

[20] Django-oscar. 2022. Should OrderDiscount.offer_id and voucher_id be ForeignKey? Retrieved Mar. 20, 2022 from https://github.com/django-oscar/django-oscar/issues/3821

[21] Edx. 2022. What Makes the Open edX Platform Unique? See These Cases. Retrieved Sep 26, 2022 from https://openedx.org/blog/what-makes-open-edx-platform-unique-see-these-cases/

[22] Edx-ecommerce. 2022. Adding E-Commerce to the Open edX Platform. Retrieved Feb 12, 2022 from https://github.com/openedx/ecommerce/tree/27e6b06b

[23] Instagram Engineering. 2016. Web Service Efficiency at Instagram with Python - Instagram Engineering. https://instagram-engineering.com/web-service-efficiency-at-instagram-with-python-4976d078e366.

[24] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 1999. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*. 213–224.

[25] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.

[26] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A data discovery system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1001–1012.

[27] Google. 2022. Google Site Reliability Engineering Book. Retrieved Mar. 25, 2022 from https://sre.google/sre-book/data-integrity/

[28] David Heinemeier Hansson. 2021. Choose a single layer of cleverness. Retrieved Feb 20, 2022 from https://dhh.dk/arc/2005_09.html

[29] Mary Jean Harrold and Mary Lou Soffa. 1994. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 2 (1994), 175–204.

[30] Arvid Heise, Jorge-Arnulfo Quiané-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. 2013. Scalable discovery of unique column combinations. *Proceedings of the VLDB Endowment* 7, 4 (2013), 301–312.

[31] IBM. 2021. Error and crash recovery from data corruption. Retrieved Nov 12, 2021 from https://www.ibm.com/docs/en/ztpf/1.1.0.15?topic=concepts-error-crash-recovery-from-data-corruption

[32] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. 2016. Verena: End-to-end integrity protection for web applications. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 895–913.

[33] Boyang Li, Denys Poshyvanyk, and Mark Grechanik. 2017. Automatically detecting integrity violations in database-centric applications. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 251–262.

[34] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*. 615–629.

[35] Xiaoxuan Liu, Shuxian Wang, Mengzhu Sun, Sharon Lee, Sicheng Pan, Joshua Wu, Cong Yan, Junwen Yang, Shan Lu, and Alvin Cheung. 2022. Leveraging Application Data Constraints to Optimize Database-Backed Web Applications. In *arXiv*. https://doi.org/10.48550/ARXIV.2205.02954

[36] Raymond A Lorie. 1977. Physical integrity in a large segmented database. *ACM Transactions on Database Systems (TODS)* 2, 1 (1977), 91–104.

[37] Philip Mayer, Michael Kirsch, and Minh Anh Le. 2017. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *Journal of Software Engineering Research and Development* 5, 1 (2017), 1–33.

[38] Jim Melton and Alan R Simon. 2001. *SQL: 1999: understanding relational language components*. Elsevier.

[39] Mongodb. 2022. Unique Constraints on Arbitrary Fields. Retrieved Mar. 25, 2022 from https://www.mongodb.com/docs/manual/tutorial/unique-constraints-on-arbitrary-fields/

[40] MySQL. 2021. How MySQL Deals with Constraints. Retrieved Nov 12, 2021 from https://dev.mysql.com/doc/refman/8.0/en/constraints.html

[41] Joseph P Near and Daniel Jackson. 2014. Derailer: interactive security analysis for web applications. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 587–598.

[42] Openedx. 2022. Open edX – Deliver Inspiring Learning Experiences On Any Scale. Retrieved Feb 12, 2022 from https://github.com/openedx/edx-platform/tree/97edc47

[43] Oracle. 2021. Oracle Database - Database Concepts - 7 Data Integrity. Retrieved Nov 12, 2021 from https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/data-integrity.html

[44] Oscar. 2022. Oscar - Domain-driven e-commerce for Django. Retrieved Feb 12, 2022 from https://github.com/django-oscar/django-oscar/tree/18c87e

[45] Oscar. 2022. Unique constraints for several table's columns. Retrieved Mar. 20, 2022 from https://github.com/django-oscar/django-oscar/pull/3868

[46] Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. 2015. Divide & conquer-based inclusion dependency discovery. *Proceedings of the VLDB Endowment* 8, 7 (2015), 774–785.

[47] Postgresql. 2021. 11.8. Partial Indexes. Retrieved Mar 12, 2022 from https://www.postgresql.org/docs/current/indexes-partial.html

[48] Postgresql. 2021. 5.4. Constraints Chapter 5. Data Definition. Retrieved Nov 12, 2021 from https://www.postgresql.org/docs/current/ddl-constraints.html

[49] Python. 2022. ast — Abstract Syntax Trees. Retrieved Feb. 12, 2022 from https://docs.python.org/3/library/ast.html

[50] Rails. 2021. Active Record Migrations. Retrieved Nov 12, 2021 from https://edgeguides.rubyonrails.org/active_record_migrations.html

[51] Rails. 2021. Concurrency and integrity for uniqueness in Rails. Retrieved Nov 12, 2021 from https://github.com/rails/rails/blob/main/activerecord/lib/active_record/validations/uniqueness.rb#L179

[52] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. 2003. *Database management systems*. Vol. 3. McGraw-Hill New York.

[53] Alexandra Rostin, Oliver Albrecht, Jana Bauckmann, Felix Naumann, and Ulf Leser. 2009. A machine learning approach to foreign key discovery.. In *WebDB*.

[54] Saleor. 2021. Error in the dashboard. Retrieved Nov 12, 2021 from https://github.com/saleor/saleor/issues/1670

[55] Saleor. 2021. Make order.total not nullable. Retrieved Nov 12, 2021 from https://github.com/saleor/saleor/pull/1893

[56] Saleor. 2021. Saleor – A headless, GraphQL-first, open-source e-commerce platform. Retrieved Feb 12, 2022 from https://github.com/saleor/saleor/tree/53e519df

[57] Saleor. 2022. Rebooting Enterprise with Open Source. Retrieved Sep 26, 2022 from https://saleor.io/enterprise-open-source/

[58] Shuup. 2022. Multivendor Marketplace Platform - Enterprise Commerce Software. Retrieved Feb 12, 2022 from https://github.com/shuup/shuup/tree/25f78c

[59] Stackexchange. 2021. Why are constraints applied in the database rather than the code? Retrieved Nov 12, 2021 from https://dba.stackexchange.com/questions/39833/why-are-constraints-applied-in-the-database-rather-than-the-code

[60] Stackoverflow. 2021. Should you enforce constraints at the database level as well as the application level? Retrieved Nov 12, 2021 from https://stackoverflow.com/questions/464042/should-you-enforce-constraints-at-the-database-level-as-well-as-the-application

[61] Teradata. 2021. Teradata - Physical Database Integrity. Retrieved Nov 12, 2021 from https://docs.teradata.com/r/sUbveBFyhttIbZzLz7nJLw/x~5k~cGbb5CIg7WrWgBNoA

[62] Gerd Wagner. 2021. Chapter 9. Implementing Constraint Validation in a Java EE Web App. Retrieved Nov 12, 2021 from https://web-engineering.info/book/WebApp1/ch09.html

[63] Wagtail. 2022. Wagtail CMS: Django Content Management System. Retrieved Feb 12, 2022 from https://github.com/wagtail/wagtail/tree/317f10

[64] Yuepeng Wang, Isil Dillig, Shuvendu K Lahiri, and William R Cook. 2017. Verifying equivalence of database-driven applications. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–29.

[65] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 286–300.

[66] Todd Warszawski and Peter Bailis. 2017. Acidrain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 5–20.

[67] Cong Yan and Alvin Cheung. 2016. Leveraging lock contention to improve OLTP application performance. *Proceedings of the VLDB Endowment* 9, 5 (2016), 444–455.

[68] Jean Yang, Travis Hance, Thomas H Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, dynamic information flow for database-backed applications. *ACM SIGPLAN Notices* 51, 6 (2016), 631–647.

[69] Junwen Yang, Utsav Sethi, Cong Yan, Alvin Cheung, and Shan Lu. 2020. Managing data constraints in database-backed web applications. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1098–1109.

[70] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M Procopiuc, and Divesh Srivastava. 2010. On multi-column foreign key discovery. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 805–814.

[71] Haochen Huang; Bingyu Shen; Li Zhong; Yuanyuan Zhou. 2023. Artifact for paper 'Protecting Data Integrity of Web Applications with Database Constraints Inferred from Application Code'. Retrieved Jan 16, 2023 from https://doi.org/10.5281/zenodo.7425382

[72] Zulip. 2021. Corrupted Reactions data model state when using multiple aliases for an emoji code. Retrieved Nov 12, 2021 from https://github.com/zulip/zulip/issues/15347

[73] Zulip. 2021. migrations: Add case-insensitive unique indexes on realm and email. Retrieved Nov 12, 2021 from https://github.com/zulip/zulip/commit/b9b146c8095648d4ef61650a89bfe6f557308574

[74] Zulip. 2021. psycopg2.errors.UniqueViolation duplicate key value violates unique constraint. Retrieved Nov 12, 2021 from https://github.com/zulip/zulip/issues/15772

[75] Zulip. 2022. Case study: Rust programming language community. Retrieved Sep 26, 2022 from https://zulip.com/case-studies/rust/

[76] Zulip. 2022. Change realm field to be not null in Attachment. Retrieved Mar. 20, 2022 from https://github.com/zulip/zulip/pull/21470

[77] Zulip. 2022. Realm field in the RealmAuditLog table. Retrieved Feb. 25, 2022 from https://chat.zulip.org/#narrow/stream/9-issues/topic/realm.20field.20in%20table%20RealmAuditLog%20and%20RealmUserDefault/near/1335322

[78] Zulip. 2022. Unique constraints for several table's columns. Retrieved Nov. 20, 2021 from https://chat.zulip.org/#narrow/stream/9-issues/topic/Several.20generated.20key.20fields.20w.2Fo.20unique%20constraints.2E

[79] Zulip. 2022. Zulip. Retrieved Feb 12, 2022 from https://github.com/zulip/zulip/tree/f5bb43ab

Received 2022-07-07; accepted 2022-09-22