Managing data constraints in database-backed web applications

Junwen Yang
University of Chicago, USA
junwen@uchicago.edu

Utsav Sethi
University of Chicago, USA
usethi@uchicago.edu

Cong Yan
University of Washington, USA
congy@cs.washington.edu

Alvin Cheung
University of California, Berkeley
USA
akcheung@cs.berkeley.edu

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

ABSTRACT

Database-backed web applications manipulate large amounts of persistent data, and such applications often contain constraints that restrict data length, data value, and other data properties. Such constraints are critical in ensuring the reliability and usability of these applications. In this paper, we present a comprehensive study on where data constraints are expressed, what they are about, how often they evolve, and how their violations are handled. The results show that developers struggle with maintaining consistent data constraints and checking them across different components and versions of their web applications, leading to various problems. Guided by our study, we developed checking tools and API enhancements that can automatically detect such problems and improve the quality of such applications.

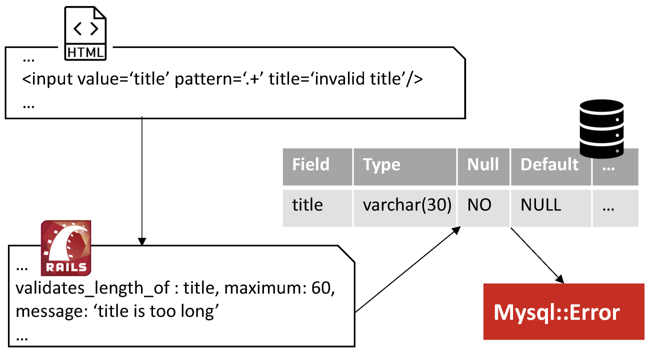

HTML
...
<input value='title' pattern='.+' title='invalid title'/>
...
Field | Type | Null | Default | ...
title | varchar(30) | NO | NULL | ...
RAILS
...
validates_length_of : title, maximum: 60,
message: 'title is too long'
...
Mysql::Error

Figure 1: Crossstack data constraints

1 INTRODUCTION

1.1 Motivation

Constraints are often associated with data used in software. These range from describing the expected length, value, uniqueness, and other properties of the stored data. Correctly specifying and checking such constraints are crucial for software reliability, maintainability, and usability. This is particularly important for database-backed web applications, where a huge amount of data generated by millions of users plays a central role in user interaction and application logic. Furthermore, such data persists in database and needs to continue serving users despite frequent software upgrades [10] and data migration [9]. As a result, consistently and comprehensively specifying data constraints, checking them, and handling constraint violations are of uttermost importance.

To better understand these challenges, consider an issue [16] reported by users of Redmine [19], a popular project-management web application written using Ruby on Rails. When a user tried to create a wiki page, she initially left the `title` field empty, which led to the "title is invalid" error message shown next to the `title` field; she then put in a long title, but got a "title is too long (maximum is 60 characters)" error; finally, she tried a title a little shorter than 60 characters, but the web page then crashed with all the filled content lost with some unreadable database error displayed.

It turned out that different constraints were specified for the `title` field across different components in Redmine. As shown in Figure 1, the front-end HTML file `views/wiki/new.html.erb` used a regular expression ".+" to specify that the title should have a positive length; the application model file `models/wiki.rb` instead used Rails `validates_length_of` API to limit the maximum title length to be 60; finally, in the database schema file `schema.rb`, the title field is declared as `varchar(30)`, limiting the maximum length to be 30.

This example illustrates how common it is for database-backed web applications to specify and check constraints for the same piece of data in different code components: the front-end browser, the application server, and the database. As such components are often separately developed and maintained, they could hold conflicting assumptions about the same piece of data. Such inconsistencies can lead to various reliability and usability problems. Particularly, a piece of data that passes all but the database checking often leads to a web-page crash, as in the above example.

Consider another example in Diaspora [22], the most popular social network application written using Ruby on Rails (according to application-stars in Github). In earlier versions, the password length is allowed to be three characters or shorter. In one version, developers decided that passwords should be longer, probably for security concerns. They then added a constraint "......+" to the password field of the log-in page, requiring the password to be at least 6 characters long. As a result, many users, whose passwords were

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380375

Table 1: Highlight results of our study
(*: all the identified issues are in latest versions of these applications)

RQ1: How are constraints specified in one software version?	
How Many?	2.1 per 100 LoC
	1.4 per 1 data field
	77% of data fields have constraints
Where?	76% in DB; 23% in application; 1% in front-end
	24% of application constraints are missing in DB
RQ2: How are constraints specified across versions?	
	49% of versions contain constraint changes
	>25% of changes tighten constraints on existing data fields
RQ3: What led to real-world constraint problems?	
Where	21% of 114 studied issues
What	51% of 114 studied issues
When	10% of 114 studied issues
How	18% of 114 studied issues
RQ4: Can we identify constraint problems in latest version?	
Where	1000+ string fields have length constraints in DB but not in app.
	200+ fields forbidden to be `null` in app. but `null` by default in DB
	88 fields required to be unique in app. but not so in DB
	57 in(ex)clusion constraints specified in app. but missed in DB
	133 conflicting length/numericality constraints between app. and DB
What	19 incorrect case-sensitivity constraints identified
How	2 missing error-message problems identified
	API default error-message enhancement preferred in user study

shorter than 6 characters, can no longer log in and are shown with the unhelpful "Please use the required format" error.

This example demonstrates that in a software world where nothing endures but change, it is challenging to make long-living persistent data endure frequent code changes, which may introduce new or even conflicting requirements to persistent data fields. Such a conflict can lead to upgrade failures, user-unfriendly error pages, and software misbehavior, like that in the above examples.

In summary, effectively managing constraints for the huge amount of persistent data in database-backed web applications (short as web applications) is critical and challenging. To understand the challenges involved, we first perform a comprehensive study to understand the specification, checking, maintenance, and violation handling of data constraints in web applications.

1.2 Contributions

In this paper, we aim to answer four key research questions about real-world database-backed web applications, as listed in Table 1 by comprehensively studying the source code, the commit history, and the issue-tracking system of 12 popular Ruby on Rails applications that represent 6 most common web-application categories.

For RQ1, we wrote scripts to collect and compare constraints expressed in various components of the latest versions of the 12 applications. We found that about three-quarter of all data fields are associated with constraints. In total, there are hundreds to over one thousand constraints explicitly specified in each application, averaging 1.1–3.6 constraints specified per 100 lines of code. Data presence and data length are the two most common types of constraints, while complicated constraints like the relationship among multiple fields also exist. We also found that hundreds to thousands of constraints specified in the database are missing in the application source code, and vice versa, which can lead to maintenance, functionality, and performance problems. The details are presented in Section 4.

For RQ2, we checked how data constraints change throughout the applications' development history. We found that about 32% of all the code changes related to data constraints is about adding new constraints or changing existing ones on data fields that have already existed in software. These changes, regardless of whether they are due to developers' earlier mistakes or warranted by new code features, can easily lead to upgrade and usage problems for data that already exists in the database. The details are in Section 5.

For RQ3, we thoroughly investigated 114 real-world issues that are related to data constraints. We categorize them into four major anti-patterns: (1) inconsistency of constraints specified at different places, which we refer to as the *Where* anti-pattern; (2) inconsistency between constraint specification and actual data usage in the application, which we refer to as the *What* anti-pattern; (3) inconsistency between data/constraints between different application versions,which we refer to as the *When* anti-pattern; and (4) problems with how constraint-checking results are delivered (i.e., unclear or missing error messages), which we refer to as the *How* anti-pattern. These four anti-patterns are all common and difficult to avoid by developers; they led to a variety of failures such as web-page crashes, silent failures, software-upgrade failures, poor user experience, etc. The details are presented in Section 6.

For RQ4, we developed tools that automatically identify many data-constraint problems in the latest versions of these 12 applications, as highlighted in Table 1. We found around 2,000 "Where" problems, including many fields that have important constraints specified in the database but not in the application or vice versa, as well as over 100 fields that have length or numericality (i.e., numerical type and value range) constraints specified in both the database and the application, but the constraints conflict with each other. We also found 19 issues in which the field is associated with case-insensitive uniqueness constraints, but are used by the application in a case-sensitive way (the "What" anti-pattern), as well as two problems related to missing error messages (the "How" anti-pattern). We manually checked around 200 randomly sampled problems and found a low false positive rate (0–10%) across different types of checks. Not to overwhelm application developers, we reported 56 of these problems to them, covering all problem categories. We received 49 confirmation from the developers (no feedback yet to the other 7 reports), among which our proposed patches for 23 of those problems have already been merged into their applications or included in the next major release.

We also developed a Ruby library that improves the default error messages of five Rails constraint-checking APIs. We performed a user study with results showing that web users overwhelmingly prefer our enhancement. The details are presented in Section 7.

Overall, this paper presents the first in-depth study of data constraint problems in web applications. Our study provides motivations and guidelines for future research to help developers better manage data constraints. We have prepared a detailed replication package for the data-constraint-issue study and the data-constraint checking tools in this paper. This package is available on the webpage of our open-source Hyperloop project [12], a project that aims to solve database-related problems in ORM applications.

Table 2: Different types of constraints in web apps

Run-time check location	Source-code location	Specification language	Specification API
Front end	View	HTML	Reg. expression
Application server	Model	Ruby	Built-in validation API
	Model	Ruby	Custom validation API
	Model/Controller	Ruby	Custom sanity check
Database server	Migration files	Ruby	ActiveRecord::Migration API
	Migration files	SQL	SQL ALTER TABLE queries

2 BACKGROUND

2.1 Architecture of web applications

Applications built using the Ruby on Rails framework are structured using the model-view-controller (MVC) architecture. For example, when a web user submit a form through a URL like `http://foo.com/wikis/new/title=release`, a *controller* action "`wikis/create`" is triggered. This action takes in the parameters from the request (e.g., "`release`" in the URL as `params[:title]`) and interacts with the database by calling the ActiveRecord API implemented by the Rails Object-Relational Mapping (ORM) framework. Rails translates ActiveRecord function calls into SQL queries (a write query in this case), whose results are then serialized into *model* objects (e.g., the `Wiki` model) and returned to the controller. The returned objects are then passed to the *view* files to generate a webpage that is sent back to users. Each model is derived from `ActiveRecord`, and is mapped to a database table by Rails. A view file (ends with `.erb` or `.haml`) usually involves multiple languages including HTML, JavaScript, and Ruby.

2.2 Constraints in web applications

We roughly categorize data constraints into three types based on where they are checked and specified as shown in Table 2.

Front-end constraints. Developers can use regular expressions to specify constraints about a particular HTML data-field inside a view file, such as the `pattern='.+'` for the `title` field in Figure 1. The majority of such constraints are related to persistent data maintained by the database.

Such constraints are checked when the user submits a web form. Failure to validate will cause the form submission to fail, with an error message specified by developers shown next to the corresponding HTML field, with all the previously filled contents remain on the page.

Application constraints. Rails developers use *validation functions* to specify constraints of data fields in model classes. Similar mechanisms exist in other ORM frameworks, such as validator functions in Django [6], and validator annotations in Hibernate [11].

A validation function is automatically triggered every time when the application saves an object of the corresponding model class (i.e., when the ORM framework saves the corresponding record into the database). Validation failure will cause the corresponding form to fail. The error message associated with the validation function will be shown to web users if developers put error checking and error-message display code in the view file.

Rails validation functions include built-in ones, which cover many common constraints like text-field lengths (i.e., `validates_length_of`, as shown in Figure 1), content uniqueness (`validates_uniqueness_of`), content presence (`validates_presence_of`), as well as custom ones, where developers express more complicated constraints like keeping a strict order among multiple fields.

Developers can also constrain a data field through custom sanity checks, although they are uncommon in Rails.

Database constraints. Many data columns are associated with constraints inside the database (DB), like the `varchar(30)` constraint shown in Figure 1. These constraints are specified in the applications' migration files, which are used to alter database schema over time. The majority of them (more than 99.5% in our studied applications) are specified through Rails Migration APIs, and are very rarely specified through SQL queries directly (<30 cases across all 12 applications we checked).

These constraints are checked by the DB when an `INSERT` or `/UPDATE` query to the corresponding columns is issued (either by the application or DB administrator). If the check fails, the application will throw an `ActiveRecord::StatementInvalid` exception to indicate an underlying DB error. Unfortunately, in practice, developers almost *never* catch such exceptions (it is caught in only 4 cases across thousands of model object saves across the 12 applications we studied). Hence, once triggered, the web user's session will most likely crash, with all the filled-in contents lost with a cryptic SQL error shown to users.

Why are the constraints distributed across components? Front-end constraints are specified for web-form input data, which is often related to DB record (e.g., used as query parameters, compared with query results, etc.). Validation functions and DB constraints are specified only for database fields, and are checked right before saving data into the DB. The expressiveness of these two are similar — most constraints that are expressible using validation functions can also be written using SQL queries or migration APIs, and vice versa. Complicated constraints expressed using custom validation functions can be expressed in the DB layer as `CONSTRAINT CHECK`s or custom stored procedures. However, neither layer can replace the other given the existence of "backdoors," e.g., DB administrators updating data using the DB console, or sharing the same DB across multiple applications. Both are common practices [4].

3 METHODOLOGY

3.1 Application selection

There are many ORM frameworks available (e.g., Ruby on Rails, Django, Hibernate, etc.). Among them, Rails is the most popular on Github. Thus, we studied 12 open-source Ruby on Rails applications, including the top two most popular Ruby applications from six major categories of web applications on GitHub: Discourse (Ds) and Lobster (Lo) are forums; Gitlab (Gi) and Redmine (Re) are collaboration pplications; Spree (Sp) and Ror ecommerce (Ro) are Ecommerce applications; Fulcrum (Fu) and Tracks (Tr) are Task-management applications; Diaspora (Da) and Onebody (On) are social network applications; OpenStreetmap (OS) and FallingFruit (FF) are map applications. All of them have been actively developed for years, with hundreds to tens of hundreds of code commits.

Table 3: # of data-constraint issues in our study and the total # of issues in the issue-tracking system

	Ds	Lo	Gi	Re	Sp	Ro	Fu	Tr	Da	On	FF	OS
Studied	14	1	16	30	31	2	1	1	11	5	0	2
Total	4607	220	18038	12117	4805	114	158	1470	3206	400	17	650

Table 4: # Data constraints in web applications

	Ds	Lo	Gi	Re	Sp	Ro	Fu	Tr	Da	On	FF	OS
DB	1403	137	1582	437	346	378	34	108	361	345	159	242
App	165	33	496	220	132	219	13	30	116	82	17	176
HTML	0	2	18	32	0	0	0	2	1	11	0	0
Total	1568	172	2096	689	478	597	47	140	478	438	176	418
LoC	62k	11k	122k	35k	31k	17k	1.7k	13k	21k	14k	7.8k	14k
#Col	1180	150	1384	338	456	384	53	107	510	268	171	228
#Col$_C$	882	104	1140	297	312	272	32	82	348	228	146	174
%Col$_C$	75%	69%	82%	88%	68%	71%	60%	77%	68%	85%	85%	76%

LoC: Lines of code. #Col: number of data columns stored in the database. #Col$_C$.: number of columns associated with constraints. Custom sanity check not considered.

3.2 Issue selection

Section 6 studies the root causes and symptoms of real-world data constraint problems using 114 reports sampled from the above 12 applications' issue-tracking systems. For the 9 applications that have medium-size issue databases (i.e., 100–5000 total reports), we randomly sampled 100 reports for each. For Redmine and Gitlab, which have more than 10,000 reports, we randomly sampled 200 reports for each. For FallingFruit, which only has 17 reports, we took all of them. Among the resulting 1317 sampled reports, we manually checked all the reports that contain keywords like "data format," "data inconsistency," "data constraint," "format change," "format conflict," etc. We finally obtained 114 reports that are truly related to data constraints, as shown in Table 3.

4 CONSTRAINTS IN ONE VERSION

To understand how many constraints are specified in software, where they are located, and what they are about, we wrote scripts to extract data constraints from the latest version of the 12 applications described in Section 3. Our scripts obtain a web application's Abstract Syntax Tree, check which Ruby validation APIs and migration APIs are used, and analyze their parameters.

In this paper, our script covers all types of constraints listed in Table 2 except for Custom sanity checks and raw SQL constraints. Both are rarely used in these applications (e.g., raw SQL constraints are only specified in fewer than 30 times across all 12 applications). Note that, when we report inconsistency or missing constraints, we manually check to make sure the inconsistency/missing constraint is not caused by our script not covering these two types of constraints.

4.1 How many constraints are there?

As shown in Table 4, there are many constraints in these applications. Across all applications, 60% - 88% of data columns are associated with constraints and there exists 1.1 to 3.6 constraint specifications for every 100 lines of code.

Summary. Data constraint specification widely exists in all types of web applications. Their consistency, maintenance, and handling affect the majority of the application data.

4.2 Where are the constraints?

As shown in Table 4, DB constraints are the most common, contributing to 58–90% of all the constraints. Application constraints contribute 10–42%, while front-end constraints are few. It is surprising that the number of DB constraints differs significantly compared to application constraints, as both are supposed to be applied to a given piece of persistent data (Section 2.2). Furthermore, inconsistencies between them can lead to application crashes as in the example shown in Figure 1. This led to the next few study items.

What DB constraints are missing in applications? Table 5 examines over 4,000 DB constraints that are missing in applications.

Alarmingly, about one quarter of these missing constraints (more than 1,000 in total) involve string/text data where developers did not specify any length constraints in the application, yet length constraints are imposed by the DB. For example, whenever creating a table column of type "string" using Rails migration API, by default, Rails framework forces a length constraint of 255-character in the database, yet many of these string fields have no length constraints specified through application validation functions. This mismatch could lead to severe problems: if a user tries to submit a long paragraph/article in such a seemingly limitless field, his application will crash due to a failed `INSERT` query, as shown in Figure 1. In fact, we found many real-world issues reporting this problem (Sec. 6.1), ultimately leading to developers adding the corresponding constraints in the application layer.

About 2% of the missing constraints, 101 in total across the 12 applications, are associated with data fields that do not exist in the application. Some of them are updated and read through external scripts, but never through the web application; others are deprecated fields that have already been removed from the application but not dropped yet from the DB. Although this does not lead to immediate software misbehavior, these cases reflect challenges in data maintenance and could cause functionality problems in the future. In addition, they cause performance problems as the database needs to maintain deprecated data.

About one third of the missing constraints are automatically satisfied by Rails or the DB and are hence benign. This includes presence and numericality constraints associated with foreign-key fields ("ForeignKey"): foreign key fields are automatically generated by Rails and satisfy presence and numericality constraints in the DB. Meanwhile, there are also constraints that are guaranteed by the DB ("SelfSatisfied"), like presence constraints guaranteed by non-null default values specified in the DB, uniqueness constraints guaranteed by an auto-increment property in the DB, etc.

The remaining one third of the constraints ("other") are difficult to analyze automatically. Based on our manual sampling and checking, most are already satisfied by how the application processes and generates corresponding data fields. Although they do not cause problems currently, developers should nonetheless be informed about them, so that code changes can be tested against these constraints to prevent regression failures.

Table 5: # Constraints in DB but not in Application

	Ds	Lo	Gi	Re	Sp	Ro	Fu	Tr	Da	On	FF	OS	All
StrLength	243	21	406	49	182	47	18	21	101	69	74	28	1259 (28%)
AbsentData	21	0	40	2	2	2	2	1	22	7	2	0	101 (2%)
ForeignKey	266	31	271	82	27	99	7	27	61	91	16	30	1008 (22%)
SelfSatisfied	192	16	161	84	28	9	2	18	3	20	8	31	572 (13%)
Others	446	39	429	126	77	89	2	29	143	82	26	64	1552 (35%)

Table 6: # Constraints in Application but not in DB
(only built-in validation constraints are listed)

	Ds	Lo	Gi	Re	Sp	Ro	Fu	Tr	Da	On	FF	OS	All
Presence	8	5	37	15	38	49	5	5	34	3	1	9	209 (51%)
Unique	3	1	12	18	19	5	0	4	16	1	0	9	88 (21%)
Inclusion/Exclusion	7	1	13	11	2	0	2	0	7	5	0	9	57 (14%)
RegEx	8	5	10	7	0	9	0	0	11	4	0	3	57 (14%)
Numeric	0	0	0	0	0	1	0	0	0	0	0	0	1 (0.2%)

False-positive analysis Besides the "Others" row in Table 5, the other 4 rows are counted by our static-checking script. To check the accuracy of our script, we randomly examined 102 cases from these 4 rows. Among these cases, we found 7 false positives: 5 are not DB constraints but are mistakenly identified due to syntax not handled by our script; 2 "StrLength" cases actually belong to "Others," as the length requirement is guaranteed by application semantics. These 102 cases include 58 "StrLength" cases, among which 5 are false positives — 3 are not DB constraints and 2 belong to "Others".

Which application constraints are not in database? Nearly 25% of the constraints specified through application validation are missing in the DB. Table 6 breaks down the ones specified through built-in validation functions based on the constraint type (412 in total). These missing constraints allow users to directly change persistent data using SQL queries in ways that are disallowed by the application, causing functionality or even security problems.[1] Furthermore, some of these missing constraints represent missed query optimization opportunities, such as improving cardinality estimation in query plan generation using such constraints [35].

About half of these missing constraints are presence constraints. That is, a field f is required to be non-null in the application, but is not required so in the DB — their default values are ironically set to be null in the DB. When users or administrators directly insert records into the DB without specifying the value for a field f, the DB would accept these records and put null into f. Subsequently, when such records are retrieved and used in the application that assumes all f to be non-null, software failures could occur.

Another category of missing constraints that can easily cause problems are uniqueness constraints. Without being specified in the DB, a uniqueness constraint often **cannot** be guaranteed by the application [13, 14]: web users could make concurrent update requests that save duplicate values into the DB, violating the uniqueness constraint and causing software failures and maintenance challenges.

[1]It is common that database administrators directly change database data using queries and scripts, bypassing the application server.

Table 7: Top 5 popular types of different layer

DB	Presence	Length	Numericality	Uniqueness	-
	1822 (32.9%)	1784 (32.3%)	1650 (29.8%)	276 (5.0%)	-
App.	Presence	Length	Uniqueness	Numericality	Inclusion
	888 (52.3%)	218 (12.8%)	209 (12.3%)	101 (5.9%)	67 (3.9%)
HTML	Presence	Length	Format	-	-
	52 (78.8%)	11 (16.7%)	3 (4.5%)	-	-

Regular expression and inclusion/exclusion constraints are rarely found in the DB layer. While these can be enforced via procedures or `ENUM` types, they are not natively supported by the Rails DB migration APIs and have to be explicitly specified via SQL, which might be a reason why they tend to be missed in the DB. Inclusion/exclusion constraints limit the value of a field to a small set of constants and would be very useful in avoiding data corruption, saving storage space, and improving database performance (e.g., through DB selectivity optimization) if they are present.

The single numeric constraint in Table 6 is a "phone number" field that is specified to be numeric in application but stored as a "string" in the database.

False-positive analysis We randomly sampled and examined 10 cases from each of the 4 main categories in Table 6 (Presence, Unique, In/Ex-clusion, RegEx). 3 out of the 40 sampled cases are false positives (2, 0, 0, 1 in the 4 categories, respectively)—syntax corner cases caused our script to identify 1 spurious presence constraint, and the remaining 2 are related to conditional constraints.

Summary. Hundreds and thousands of database constraints do not exist in application, and vice versa. The majority of these discrepancies can actually lead to bad user experience (missing string length constraints), database maintenance challenges (data fields that are no longer used in the application), code maintenance challenges (constraints implicitly guaranteed by the application logic), data corruptions, software failures, or sub-optimal database performance (missing DB constraints). They can be avoided by implementing constraints in the application and as SQL constraints in the database. However, in practice inconsistencies are likely inevitable if we only rely on developers' manual effort. It would be helpful to develop automated techniques that coordinate database and application constraints.

4.3 What types of constraints are there?

Standard types. Table 7 shows the most popular constraint types among all front-end, application built-in validation, and DB constraints. The top 2 most popular types are consistently presence and length.

Custom validation constraint types. Custom validation functions are used much less often than built-in ones, but are not rare, contributing about 5% to slightly over 25% of all application validation functions across the 12 applications (avg. 18% across all apps). We randomly sampled 50 custom validation functions and found that more than half of them are used to check multiple fields at the same time (27 out of 50), like the function `presence_of_content`

Table 8: App. versions with constraint changes (#Version$_C$)

	Ds	Lo	Gi	Re	Sp	Ro	Fu	Tr	Da	On	FF	OS
#Version	316	19	1040	159	253	31	7	26	39	86	12	95
#Version$_C$	187	18	563	44	89	20	4	12	26	18	10	41
%Version$_C$	59%	95%	54%	28%	35%	65%	57%	46%	67%	21%	83%	43%

All types in Table 2 except for custom sanity checks are considered.
Red apps use a release as a version; Black apps use every 100 commits as a version.

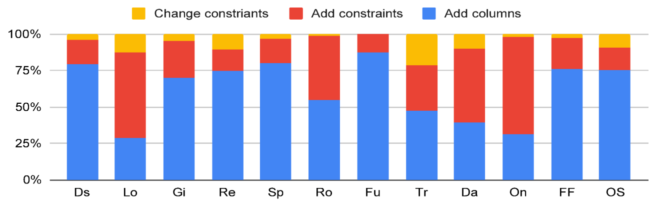


Figure 2: Breakdown of # of adding/changing constraints

in the `StatusMessage` model from `Diaspora`, which requires that at least one of the fields `text` or `photos` be non-empty. These custom validations seldom have corresponding constraints in DB — only 4 out of the 50 we sampled exist in DB.

Custom sanity check types. We also sampled 20 sanity checks on input parameters from the controller code of 5 applications. Among these 20 checks, the majority (17) are indeed checking inputs that are related to persistent data stored in the DB. Among these 17, 5 are about data constraints, including presence and inclusion constraints, while the others are related to conditional data update/re-processing. Among these 5 constraints, only 1 is specified in an application validation function and none exists in the DB.

Summary. Although simple constraints like presence, length, and numericality are the most common, more complicated constraints, such as those involving multiple fields, are also widely used. Most of the custom constraints are missing from the DB, while constraints reflected by sanity checks are often missing in both the application and DB. Future research that can automatically reason about custom sanity checks and custom validation functions can greatly help to identify and add missing constraints.

5 CONSTRAINTS ACROSS VERSIONS

The software maintenance task for web applications comes with the extra burden of database maintenance, including both data format changes, like adding or deleting a table column, and data constraint changes, like changing the length requirement of a password field. In this section, we study how constraints evolve across versions and the related data-maintenance challenges.

How often do constraint-related changes occur? We first checked the first commit of each application, and found *no* data constraints in all but 3 applications (Ds, Lo, FF). For all applications, the majority of the constraints were added in later commits.

As shown in Table 8, 21–95% (avg. 49% across all apps) of code versions contain data constraints that are different from those in its previous version, indicating that constraint changes are common. Note that, for most applications, we treat one code release as one version; for 4 applications that do not specify release/version information, we treat every 100 code commits as one version.

What triggered changes? We categorize all the cross-version changes about DB constraints and application validation constraints into three types: (1) Add Column: adding constraints to a data column that did not exist in previous version; (2) Add Constraint: adding constraints to an existing data column that was not associated with that specific type of constraints, like adding a length constraint to a data field that had no length constraint previously; (3) Change Constraint: changing the detailed requirement of a constraint that already existed in previous version.

What is alarming from the result (Figure 2) is that the Add-Column type only contributes to around or lower than 50% of changes in 5 out of the 12 applications. On the other hand, 13–67% of constraint changes (23% across all applications) are adding new types of constraints to columns that already existed in earlier versions (i.e., tightening the constraints), indicating that constraint addition is often developers' after-thoughts. Changing existing constraints is much less common, but is still not rare, contributing to more than 10% of constraint changes in 4 applications.

Summary. It is problematic that around or more than a quarter of constraint-related code changes in most applications are about adding constraints to already existing data columns. This indicates a widely existing vulnerability that allows constraint-violating data to be stored into the database before the correct constraints are imposed. Tools are needed to help developers add suitable constraints whenever a new data column is created and warn of data that is incompatible with the newly added constraints.

6 CONSTRAINT-RELATED ISSUES

We categorize 114 real-world issues into 4 types as shown in Table 9.

6.1 WHERE is the constraint specified?

As discussed, application and DB constraints for the same data field can be inconsistent with each other. Such inconsistencies contributed to 24 out of the 114 issues.

Application constraints looser than DB constraints. 13 out of 24 issues fall into this category. In 12 of them, the constraint is completely missing in application layer while the rest one issue happens because the length constraint has a smaller value in database layer than in application layer. In these cases, a record saving operation would pass the application server's checking but fail in the DB, causing a web page crash with an unhandled raw DB error thrown to end users, which is often difficult to understand and causes poor user experience. The example discussed in Figure 1 is an illustration.

Application constraints stricter than DB constraints. 11 out of 24 issues fall into this category. In 9 of them, application constraints are not defined in database layer at all while the rest two are caused by that length constraint has smaller value in application layer than in database layer. In these cases, the application misbehaves as the administrator/user directly changes database records through SQL queries in a way that violates application constraints. This happens quite often. For example, Spree [20], an

Table 9: Data-constraint issues in real-world apps

		Ds	Lo	Gi	Re	Sp	Ro	Fu	Tr	Da	On	FF	OSM	SUM
WHERE		3	0	3	7	8	0	1	0	0	2	0	0	24
WHAT	vs. code	0	1	8	11	14	1	0	0	4	2	0	0	41
	vs. user	6	0	0	4	3	1	0	1	1	1	0	0	17
WHEN		3	0	4	1	3	0	0	0	1	0	0	0	12
HOW		2	0	1	7	3	0	0	0	5	0	0	2	20
SUM		14	1	16	30	31	2	1	1	11	5	0	2	114

```
Error: undefined method `image' for nil:NilClass in line 2
```

```
1 order.line_items.each do |item|
2  item.variant.image
3 end
```

(a) orders/_form.html.erb

```
class LineItem
validates_presence_of :variant, :order
end
```

(b) constraint in line_item.rb

Figure 3: Constraint mismatch in Spree

on-line shopping system, has 4 issues caused by administrators modifying database content through direct SQL requests. Discourse [2] even has scripts that bypass model constraints to import other forum applications' data.

Figure 3 shows such an issue [21] in Spree. As shown in (b), each `LineItem` is associated with a `variant` object and a `presence` constraint is used to ensure the existence of every associated `variant`. This ensures that an expression like `item.variant.image` in (a) is never null. However, this constraint does not exist in the database. In this bug report, an adminstrator accidentally deleted a `variant` record in the DB that is associated with a `LineItem` record, and that led to a null pointer error when he tried to display an order through the code in Figure 3a.

Summary. As shown by real-world issues, inconsistencies between application and database constraints cause problems, including web page crashes and poor user experience. Considering the hundreds and thousands of constraints that exist in the DB but not in application and vice versa (see Section 4.2), this problem could be much more severe and widespread than what reflected by the issue reports. Automatically detecting such constraint inconsistencies will be very helpful, which we further explore in Section 7.

6.2 WHAT is the constraint about?

The most common problem is a mismatch between how data is supposed to be used in the application and the constraints imposed on it. This accounts for 58 out of 114 issues.

6.2.1 *Conflict with user needs.*

Users sometimes would relax an existing constraint, such as increasing the input length of name field in tracker from 30 to 100 (Redmine-23235 [17]). These contribute to about 10% of the issues in our study. Developers usually satisfy the users' desires and change constraints accordingly.

Summary. For certain type of constraints, like the length constraint, it is difficult to have one setting that satisfies all users' needs. It would be helpful if refactoring routines can be designed to turn a fixed-setting constraint into configurable.

```
  errors.add(:value, :not_a_date) unless
  value =~ /^\d{4}-\d{2}-\d{2}$/
+ && begin; value.to_date; rescue; false end
```

Figure 4: Type conflict example in Redmine

```
class User
validates_uniqueness_of :email, :case_sensitive => false
```

```
- user = User.where(username: username)
+ user = User.where("lower(name) = ?", name.downcase).first
```

Figure 5: Case sensitivity conflict example in Gitlab

6.2.2 *Conflict with application needs.*

Many constraints are created to guarantee program invariants that are crucial to applications' functional correctness. Constraints that are insufficient or even conflicting with how the corresponding data is used by the application contribute to more than one third of all the issues in our study.

Type conflicts. These constraints treat a data field as having a general type, but the application uses the data field in a more specialized way that demands tighter constraints. In one Redmine issue [18], a user noticed that she can input invalid dates like "2011-10-33" without triggering any errors. This problem happened because Redmine only used a regular expression "\d{4}-\d{2}-\d{2}$" to make sure the input follows the "yyyy-mm-dd" format without more detailed checking. To solve this problem, Redmine later added "value.to_date" to check whether the input can really be converted to a date or not in the custom validate function shown in Figure 4.

Case sensitivity conflicts. *Uniqueness* is a common constraint associated with a data field to avoid duplication, like preventing two users from having the same ID. A common problem is that a field is written to the DB in a case-sensitive way of uniqueness, while used or searched in a case-insensitive way, or vice versa. Such inconsistency can lead to severe software misbehavior.

In a Gitlab issue [7], a user's profile email is all in lower case, but she committed code with an upper-case letter in her email, which then cannot be matched to her profile. What is annoying was that she was unable to add the different casing as an alias, as Gitlab said the email was "already in use." This happens because when a user-email is stored into DB, the uniqueness checking is case insensitive—"abc@example.com" is treated the same as "ABC@example.com." However, when the application searches for code commit using email as the index, the search is case sensitive — code committed by "ABC@example.com" cannot be retrieved by a search using "abc@example.com." The patch made the search also case insensitive, thus always converting the input email to pure-lowercase before the search, as shown in Figure 5.

Boundary value conflicts. There are cases where certain values of a data field are allowed by the application logic, but disallowed by the constraints. For example, in the typical checkout flow of Spree, users would enter their delivery details, then proceed to a payments page to enter discounts and payment details, and then finally arrive at a confirmation page. However, in one Spree issue[23], a user complained that when she entered a discount coupon that reduced the price to zero—which was actually a valid use case—the application did not allow him to proceed, and instead redirected back to the delivery page. The source of the bug was a constraint in the model layer (`models/spree/order.rb`) which incorrectly required the value of the `total` field to be strictly greater than zero.

Summary. Failure symptoms of these bugs are quite different from all the other types of bugs (WHERE, WHEN, HOW). They can lead to severe software misbehavior or even disable an entire feature of a web application. It would be ideal if a program analysis tool can compare how a data field is used in software and identify inconsistency between how it is used and how it is constrained. This is challenging for generic data types and data usage, but is feasible for specific types of problems, which we explore in Sec. 7.

6.3 WHEN is the constraint created?

When upgrading an application, sometimes newly added or changed constraints might be incompatible with old data. 12 issues are caused by such inconsistency across versions. The failure symptoms vary based on the different program context where the tighter constraint is checked.

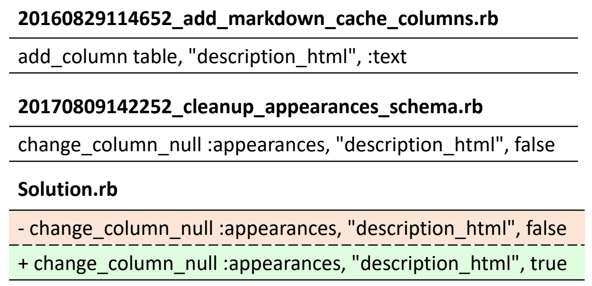
Read path. When a constraint is newly created or tightened along a DB-record loading code path (e.g., front-end constraint or application sanity-check changes), an incompatible new constraint can cause failures in loading old data and hence severe functionality problems. The Diaspora example in Section 1 belongs to this category: the password's length requirement tightened and hence invalidated many old passwords.

Write path. When a constraint is newly created or tightened along a path that intends to save a record to the database (e.g., all the application-validation constraints and database constraints), the incompatibility between the new constraint and old data can be triggered under the following two circumstances.

First, all the old data in the database will be checked against the new set of DB constraints during a migration process during application upgrade. Inconsistency between old data and new DB constraints can cause an upgrade failure. For example, one Gitlab issue [8] complains that they failed to upgrade from version 9.4.5 to 9.5.0 due to `NotNullViolation` during data migration. As shown in Figure 6, the `decription_html` column was added to Gitlab before version 9.4.5 (in the "20160829..." migration file) and was filled with `nulls` by default.[2] Later on, in the "20170809..." migration (shown in the Figure 6), a non-null constraint was added to the column through the API `change_column_null` with parameter `false`. This caused many users' upgrade to fail because there were many old records with a default `null` in that column. The patch removed the "non null" constraint to the `description_html` column, as shown in Figure 6.

Second, even if all the old data is validated against DB constraints and the application has successfully upgraded, the old data might still conflict with new constraints specified through the application validation APIs that did not exist in the prior version. This can lead to problems when the application allows users to edit an existing record—users may have trouble in saving an edited record back. In one Discourse issue [3], a user complained that she made a small edit to an old post's title, but was unable to save with an error

[2]When no default value is specified in `add_column`, `null` is used as the default value.



```
20160829114652_add_markdown_cache_columns.rb
add_column table, "description_html", :text

20170809142252_cleanup_appearances_schema.rb
change_column_null :appearances, "description_html", false

Solution.rb
- change_column_null :appearances, "description_html", false
+ change_column_null :appearances, "description_html", true
```

Figure 6: Old data conflicts with new constraints in Gitlab

message stating that the title was invalid. It turned out that, the title's length constraint has been changed from 30 to 20 characters in the application's validation function. That old post's title contained 28 characters; the small edit did not change the title length. So, the old post can still be loaded by the application, but cannot be saved back after such small edits.

Summary. Given the frequent constraint addition and changing in web applications, it is inevitable that old data may become incompatible with new constraints. It would be helpful if automated tools can provide warnings for developers when constraints become tighter in a new version, particularly (1) if the migration file has high probability to fail (e.g., specifying a constraint that conflicts with a column's default value), then developers should fix the migration file; (2) if the application allows editing old data, then developers should probably add explicit warning to users about the risk of editing old data; and finally (3) the case of having tighter constraints that limit the reading of old data should be avoided. We explore this in Section 7.

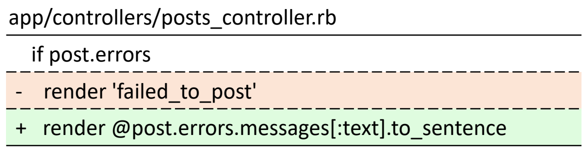
6.4 HOW are the checking results delivered?

Constraint violation is common in web applications, as web users cannot anticipate all the constraints in advance and will inevitably input constraint-violating data. Consequently, delivering informative and friendly error messages is crucial to web applications' user experience. 20 issues in our study are about this problem.

These 20 issues are mostly related to application-validation constraints. Rails validation APIs provide default error messages that are mostly clear.[3] However, developers sometimes forgot to display the error message associated with the validation APIs (8 cases) and sometimes override the default message with uninformative generic messages (12 cases), which led to user complaints. For example, in a Diaspora issue [1], a user complained that when he tried to post a long article, the posting failed with an unhelpful error message "Failed to post!" Developers found out that their code in `posts_controller` forgot to render the error message defined in `post`'s validation function. The patch fixed this problem and would display the required length limit, as shown in Figure 7.

Summary. Developers should be reminded to display error messages associated with validation APIs. Future IDEs should automatically synthesize default error checking and error-message display code. Improving the quality of default and custom error message is crucial to user experience. We will explore this in Section 7.

[3]Section 7 discusses cases when the default message is unclear and how we enhance it.



```
app/controllers/posts_controller.rb
  if post.errors
-   render 'failed_to_post'
+   render @post.errors.messages[:text].to_sentence
```

Figure 7: Unclear error message in Diaspora

7 SOLUTIONS & EVALUATION

We now discuss our experience in building tools to automatically discover the anti-patterns discussed earlier. We focus on applying them to the latest versions of the studied applications, as these represent potential bugs that have not been discovered.

7.1 Where issues

As discussed in Section 4.2, our scripts can automatically find more than 1000 string-length DB constraints that are missing in application, and more than 400 application built-in-validation constraints that are missing in the DB. We reported 16 of them covering different types, with 12 of them already confirmed by developers from 3 applications (Lo, Ds, FF).

In addition, we extended our scripts to automatically find conflicting cases, where the same type of constraint, like length, is specified for the same data field in both database and application, but the exact constraint requirement is different.

As shown in Table 10, our checker reported 138 conflicting constraints in total. Our manual checking confirmed that 133 of them are true conflicts and 5 are false positives.

These 133 conflicts include 84 cases where applications' length constraints are tighter than the DB's, 4 cases in the other way, 1 case where the columns referenced by uniqueness constraints did not exactly match, and 44 cases where the range or type of numeric values allowed in DB did not match the corresponding restriction in the model. For example, our results showed that, in the Tracks application, there was a string field `description` in model `Todo` for which the length in DB was limited to 255 characters, but was limited to 300 in the model. We reported this mismatch to developers and received confirmation that it was indeed a bug. As another example, we found 5 instances in OpenStreetMap where developers meant to require fields to be integers in both the DB and application. However, developers had typos in their use of validation APIs, which caused the application-level numericality constraints to be silently ignored. We reported this to developers, who then fixed the bug.

As an example of range mismatch, there was a case in Spree where the field `price` must be greater than or equal to zero. However, in the DB, the field type was `decimal` which allows negative values.

Among the 5 false positives, 3 were caused by our tool's limited ability in handling non-literal expressions, and the others were related to our tool's inability to distinguish between array length and string length validations.

7.2 What issues

We built a checker to detect "case-sensitivity conflicts" discussed in Section 6.2.2. Our checker first identifies every field that has case insensitive constraints specified by the validation API `validates_uniqueness_of:field` and `case_sensitive:false`, then checks all the statements that issue a read query to load such a field to see if the loading is ever done in a case sensitive way. To identify all those read queries, we used an existing static analysis framework for Rails [48]; to identify case-sensitive loading, we check whether the query is directly ordered by the field (`.order('field')`) or filtered on the field (`.where(field: params)`) without case conversion.

Our checker found 19 issues in latest versions — 14 in Lobsters, 3 in Redmine, 2 in Tracks. Our manual checking confirmed these are all bugs (no false positives). We also got confirmation from developers of Lobsters and Redmine. Redmine has already added our patch to their next major release 4.1.0.

7.3 When issues

Given two code versions, to detect inconsistency between old data and new constraints, we extend our script that examines constraint changes across versions (Section 5) to see if new constraints are added or existing constraints are tightened. We then further check whether the application allows editing existing DB data, whether the default value conflicts with the new/changed constraint, and whether the migration file updates the corresponding column in the database, which is a common way to avoid incompatibility problems. Due to space constraints, we omit details of the algorithm.

We applied our checker to the 12 applications. It did not find problems with the latest upgrade of these applications.

7.4 How issues

Improving built-in error messages. Rails built-in validation APIs provide default error messages that are used by developers in most cases, only overridden in 2% of the cases across all studied applications. Consequently, having informative default messages is crucial.

Table 10: # Mismatch constraints between DB-Model

	Ds	Lo	Gi	Re	Sp	Ro	Fu	Tr	Da	On	FF	OS
Length - DB looser	5	7	12	9	0	25	0	4	4	11	0	7
Length - DB tighter	0	0	0	0	0	3	0	1	0	0	0	0
Uniqueness	1	0	0	0	0	0	0	0	0	0	0	0
Numericality	4	0	24	1	6	0	3	0	0	0	3	3
False positives	0	0	2	3	0	0	0	0	0	0	0	0
Total	10	7	38	13	6	28	3	5	4	11	3	10

Table 11: Our enhancement to default error messages

	Default	Enhanced
`inclusion_of`	"invalid"	"have to take values from {A, B, ...}"
`exclusion_of`	"reserved"	"cannot take values from {A, B, ...}"
`confirmation_of`	"invalid"	"Case does not match with earlier input"
`uniqueness_of`	"invalid"	"Not unique in case (in)sensitive comparison"
`associated`	"o is invalid"	"field f of object o is invalid"

Table 12: User study results

Task-1	# input attempts w/ modified	# attempts w/ default	Decrease
Inclusion	2.2	3.1	30%
Associated	2.3	3.4	33%

Task-2	% of users prefer modified	% prefer default	No preference
Exclusion	74%	22%	4%
Confirmation	81%	8%	11%
Uniqueness	74%	16%	10%

We found that 5 APIs' default messages can be more informative, as shown in Table 11. For example, `validates_confirmation_of` ensures that a field and its confirmation field have the same content. Instead of only saying the input is "invalid," we add information on whether the matching failure is caused by case sensitivity, so the user can decide whether to change just the case or the actual value. As another example, `validates_associated` checks if every field of a sub-object *o*, which is associated with another object, is valid (e.g., a "photo" is a nested object of "profile", and has fields "source_url", "width", "height"). If the validation of any field of *o* fails, the default message states only that the entire *o* is invalid. Our enhancement lets the user know which specific field (e.g., "source_url" or "width" or "height") is incorrect and how to revise.

We have implemented a library (i.e., a Ruby gem) to overwrite the Rails default error message with our advanced ones. Our gem redefined the existing error message generation functions with custom ones that incorporated more information.

User Study. To evaluate our error-message changes, we recruited 100 participants using Amazon Mechanical Turk (MTurk). The participants are all live in US and are at least 18 years old with higher than 95% MTurk Task Approval rate. We asked users to perform two tasks. First, users provided answers to questions such as, enter a title, first name, and last name; or try and enter a unique value for a given category. If they fail to provide a valid answer, we either provide them with the Rails default error message, or our improved error message. In each case, we track the number of retries required for the user to reach a valid input, and if they cannot after 5 retries, we skip to the next question. Each user was given 2 of these tasks. In the second task, we provide a webpage screenshot of a question and an incorrect answer-input to that question. The questions are based on the applications we studied. We then show two options for error messages: the default message and the improved message. We ask the user to rate which error message would be more helpful in arriving at a valid input. Each user was given 3 of these tasks.

As shown in Table 4, for the first task, our enhanced error messages reduced the number of tries users took to reach valid inputs by about 30%; for the second task, we find 74–81% of users preferred our enhanced error messages, depending on the type of validation.

Detecting missed error messages. Developers are required to provide error messages for custom validations through the API `object.errors.add(msg)`. We extend our script that identifies custom validation functions to further check if an error message is provided. We found one case in Diaspora where the error message is missing. This is actually a severe problem: since Rails uses the count of error messages to determine the validity of an object, an invalid object can then be incorrectly treated as valid and lead to application failures. We reported this bug to Diaspora developers, who have confirmed that this is indeed a bug.

Detecting missed error rendering. Since there are many ways to render error messages on a web page, it is difficult to automatically detect this problem. We randomly chose 45 HTML pages with forms across 12 applications, and manually checked if error messages caused by invalid inputs were rendered. We found one case where the message would never be rendered: on a page in OpenStreetMap that asks users to input a URL, when the input has an improper format, the web page marks the field with red color, without rendering the error message associated with the constraint.

8 DISCUSSION

8.1 Impact of False Positives

Our scripts for checking constraints inconsistency across layers has some false positives, of which the vast majority come from two types of constraints: (1) string-length constraints in database, (2) presence constraints in applications. The remaining false positives are due to some validation/migration API call parameters being derived from function calls or non-constant expressions, which we do not currently evaluate.

Such false positives have limited impact on the paper's findings and are already considered in our finding presentation:

RQ1: This has little impact. The overall trends like many data fields associated with constraints, DB containing most constraints will not be affected by these small number of false positives.

RQ2: This has negligible impact. For instance, the number of versions with constraint changes remains the same even if we do not consider the above two types of constraints;

RQ3: There is no impact since the real-world issue study is conducted manually;

RQ4: This has negligible impact. All findings in Table 1 still hold, as they either are not related to those two types of constraints or are reported with false positives already pruned or carefully considered. For instance, although our script reported 1,650 database string length constraints missing in the application, we intentionally only highlight "1000+ string fields ...", instead of 1,650, in Table 1, exactly because we have taken the potential impact of false positives into account.

8.2 Threats to Validity

Internal Threats to Validity: As discussed in Section 2.2 and 4, we only considered DB constraints declared through Rails built-in migration APIs, but not those through SQL queries, which are extremely rare (fewer than 30 across all 12 applications). Our analysis covers only native DB types such as string, numeric, and datetime types, and excludes non-native DB types such as JSON, spatial, or IP format, which together account for less than 1% of all columns. Front-end constraints specified through JavaScript files were not considered. Finally, our static checkers have false positives as discussed in Section 4.2 and 7.

External Threats to Validity: The 12 applications in our study clearly may not represent all real-world applications; the 114 issues studied also may not represent all constraint-related issues in these applications; the 100 participants of our user-study from MTurk may not represent all real-world users. Overall, we have tried our best to conduct an unbiased study.

As discussed in Section 2.2, other ORM frameworks, like Django and Hibernate, also let developers specify application and database constraints like that in Rails. We sampled 22 constraint-related issue reports from the top 3 popular Django applications on Github, and observed similar distributions, as shown below.

	WHERE	WHAT vs.code	WHAT vs.user	WHEN	HOW	SUM
django-cms [5]	1	2	3	3	1	10
zulip [24]	1	4	2	0	0	7
redash [15]	0	2	0	0	3	5

9 RELATED WORK

Verifying data constraints. Prior work has investigated verifying database-related constraints. ADSL [26] verifies data-model related invariants (e.g., whether each todo object is associated with a project object) using first order logic, while the invariants are provided by users using their invariant language. Singh and Wang [39, 42] check whether a set of DB constraints still hold when DB schema evolves while Caruccio [27] conducts a survey of related work in this domain. Pan [37] proposes a method to leverage symbolic execution to synthesize a database to verify different types of constraints like query construction constraints, DB schema constraints, query-result-manipulation constraints, etc.

Verifying web applications using constraints. Another line of work focuses on using constraints provided by the DB or application for application verification and synthesis, like verifying the equivalence of two SQL queries[30, 31, 40], DB applications [41], synthesizing a new DB program with a new scheme given the original program with an old scheme [42], and handling chains of interactive actions [32].

Other types of data constraints. Much previous research has looked at how to specify security/privacy-related data constraints and how to verify or enforce those constraints across different components of database-backed applications [25, 33, 36, 46]. These constraints are currently not supported by web application frameworks, and are orthogonal to this study.

Leveraging constraints to improve performance. Using database constraints to improve query performance is already widely adopted in database systems. For example, Wang [34] leverages foreign key constraints to accelerate the sampling of join queries. Other work leverages DB data constraints to find an equivalent but more efficient query plan, for instance, Chestnut [44] adds constraints as extra assumptions to help synthesize better query plans, and Quro [43] leverages data access constraints to optimize transactional applications. Although much work looked at how to leverage data constraints, little work has been done on studying how the constraints are defined and used in DB-backed applications, or what are the common issues related to these data constraints. Our work reveals that developers are spending a lot of effort managing constraints and suffer many problems that are hardly paid attention to in research work. These findings open new research opportunities like automating constraint-consistency check or making the constraint changes easier for developers.

Empirical study of web applications. Past empirical studies looked at different aspects of web applications, like ORM-related performance problems [28, 29, 45, 47] and client-side performance problems [38] but not data-constraint problems.

10 CONCLUSION

Specifying and maintaining consistent and suitable constraints for data is crucial to ensure the application correctness and usability. In this paper, we thoroughly studied how data constraints have been specified, maintained, and led to real-world issues in 12 representative open-source DB-backed applications. Our study shows that tooling support is needed to help developers manage data constraints, and our checker is the first step towards providing such support.

ACKNOWLEDGEMENT

This work is supported in part by the NSF through grants CCF-1837120, CNS-1764039, 1563956, 1514256, IIS-1546543, 1546083, 1651489, OAC-1739419, and the computation resource from Chameleon cloud; DARPA award FA8750-16-2-0032; DOE award DE-SC0016260; the Intel-NSF CAPA center; gifts from Adobe, Google, and CERES research center for Unstoppable Computing. We thank Madan Musuvathi and Suman Nath from Microsoft Research for inspiring us to start this research direction.

REFERENCES

[1] *Diaspora-5090.* https://github.com/diaspora/diaspora/issues/5090.
[2] *Discourse.* A blog application. https://github.com/discourse/discourse/.
[3] *Discourse-89148.* https://meta.discourse.org/t/89148.
[4] *Discourse Import Scripts.* A blog application. https://github.com/discourse/discourse/tree/master/script/import_scripts.
[5] *Django-cms.* An enterprise content management system. https://github.com/divio/django-cms/.
[6] *Django Validator Function.* https://docs.djangoproject.com/en/2.2/ref/validators/.
[7] *Gitlab-24493.* https://gitlab.com/gitlab-org/gitlab-ce/issues/24493.
[8] *Gitlab-36919.* https://gitlab.com/gitlab-org/gitlab-ce/issues/36919.
[9] *Gitlab database migrate.* https://github.com/gitlabhq/gitlabhq/tree/master/db/migrate.
[10] *Gitlab releases.* https://about.gitlab.com/releases/.
[11] *Hibernate Validator Annotation.* https://hibernate.org/validator/documentation/getting-started/.
[12] *Hyperloop.* https://hyperloop-rails.github.io/vibranium/.
[13] *Rails Uniqueness API.* https://github.com/rails/rails/blob/master/activerecord/lib/active_record/validations/uniqueness.rb#L165/.
[14] *Rails Uniqueness Problem.* https://thoughtbot.com/blog/the-perils-of-uniqueness-validations.
[15] *Redash.* An application to connect your company's data. https://github.com/getredash/redash/.
[16] *redmine-24283.* https://www.redmine.org/issues/24283.
[17] *Redmine-25235.* http://www.redmine.org/issues/25235/.
[18] *Redmine-9394.* http://www.redmine.org/issues/9394/.
[19] redmine, a project management application. https://redmine.org/. (????).
[20] *Spree.* A ecommerce application. https://github.com/spree/spree/.
[21] *Spree-3829.* https://github.com/spree/spree/issues/3829.
[22] *Spree-4123.* https://github.com/diaspora/diaspora/issues/4123.
[23] *Spree-6673.* https://github.com/spree/spree/issues/6673.
[24] *Zulip.* A powerful team chat system. https://github.com/zulip/zulip/.
[25] Muath Alkhalaf, Shauvik Roy Choudhary, Mattia Fazzini, Tevfik Bultan, Alessandro Orso, and Christopher Kruegel. 2012. Viewpoints: differential string analysis for discovering client-and server-side input validation inconsistencies. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 56–66.
[26] Ivan Bocić, Tevfik Bultan, and Nicolás Rosner. 2019. Inductive verification of data model invariants in web applications using first-order logic. *Automated Software Engineering* 26, 2 (2019), 379–416.
[27] Loredana Caruccio, Giuseppe Polese, and Genoveffa Tortora. 2016. Synchronization of queries and views upon schema evolutions: A survey. *ACM Transactions on Database Systems (TODS)* 41, 2 (2016), 9.
[28] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting Performance Anti-patterns for Applications Developed Using Object-relational Mapping. In *ICSE*. 1001–1012.
[29] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2016. Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks.. In *ICSE*. 1148–1161.
[30] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *CIDR*.
[31] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: Proving Query Rewrites with Univalent SQL Semantics. In *PLDI*. 510–524.
[32] Alin Deutsch, Monica Marcus, Liying Sui, Victor Vianu, and Dayou Zhou. 2005. A verifier for interactive, data-driven web applications. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 539–550.
[33] 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.
[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*. ACM, 615–629.
[35] Guy Lohman. *Is Query Optimization a "Solved" Problem?* https://wp.sigmod.org/?p=1075.
[36] 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*. ACM, 587–598.
[37] Kai Pan, Xintao Wu, and Tao Xie. 2014. Guided test generation for database applications via synthesized database interactions. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 2 (2014), 12.
[38] Marija Selakovic and Michael Pradel. 2016. Performance issues and optimizations in javascript: an empirical study. In *ICSE*. 61–72.
[39] Rohit Singh, Vamsi Meduri, Ahmed Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. 2017. Generating concise entity matching rules. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1635–1638.
[40] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2018. Speeding up symbolic reasoning for relational queries. *PACMPL* 2, OOPSLA (2018), 157:1–157:25.
[41] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. 2017. Verifying Equivalence of Database-driven Applications. In *Proceedings of the ACM on Programming Languages*. 56:1–56:29.
[42] 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*. ACM, 286–300.
[43] Cong Yan and Alvin Cheung. Leveraging Lock Contention to Improve OLTP Application Performance. *Proc. VLDB Endow. (2016)*, 444–455.
[44] Cong Yan and Alvin Cheung. 2019. Generating Application-Specific Data Layouts for In-memory Databases. *Proc. VLDB Endow.* (2019), 1513–1525.
[45] Cong Yan, Junwen Yang, Alvin Cheung, and Shan Lu. 2017. Understanding Database Performance Inefficiencies in Real-world Web Applications. In *CIKM*.
[46] 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.
[47] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *ICSE*. 800–810.
[48] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. PowerStation: Automatically detecting and fixing inefficiencies of database-backed web applications in IDE. In *FSE*. 884–887.