THE UNIVERSITY OF CHICAGO

IMPROVING PERFORMANCE AND CORRECTNESS OF DATABASE-BACKED WEB APPLICATIONS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE
IN CANDIDACY FOR THE DEGREE OF
PHILOSOPHY OF DOCTOR

DEPARTMENT OF COMPUTER SCIENCE

BY
JUNWEN YANG

CHICAGO, ILLINOIS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

ABSTRACT

Modern web applications have to manage a lot of user data and often use a three-stack architecture: (1) a web interface developed by markup language like HTML (2) application logic developed by traditional object-oriented languages, and (3) a database management system (DBMS) that maintains persistent data.

Under this architecture, there are common understanding gaps between web designer and application developers and database engine. As a result, performance and correctness problems are common.

To tackle performance problems, we did one of the first studies to understand why real-world web applications are slow. Guided by that study, we used cross-stack analysis to synthesize efficient web pages, to automatically detect inefficiency in data processing code, and to help database optimization using application knowledge. Our tools have found thousands of performance issues. To tackle correctness problems in web applications, we investigated how data constraints could be inconsistent across web interface, application, and database, and how to solve these problems.

CHAPTER 1
INTRODUCTION

Modern web applications face the challenge of processing a growing amount of data while serving increasingly impatient users. On one hand, popular web applications typically increase their user bases by 5–7% per week in the first few years [67], with quickly growing data that is produced or consumed by these users and is managed by applications. On the other hand, studies have shown that nearly half of the users expect a site to load in less than 2 seconds and will abandon a site if it is not loaded within 3 seconds [49], while every extra 0.5 second of latency reduces the overall traffic by 20% [73].

To manage large amounts of data, modern web applications often follow a three-stack architecture: (1) a web interface developed by markup language like HTML (2) application logic developed by traditional object-oriented languages, and (3) a database management system (DBMS) that maintains persistent data. To help developers construct such database-backed web applications, Object Relational Mapping (ORM) frameworks have become increasingly popular, with implementations in all common general-purpose languages: the Ruby on Rails framework (Rails) for Ruby [42], Django for Python [16], and Hibernate for Java [24]. These ORM frameworks allow developers to program such database-backed web applications in a DBMS oblivious way, as the frameworks expose APIs for developers to operate persistent data stored in the DBMS as if they are regular heap objects, with regular-looking method calls transparently translated to SQL queries by frameworks when executed.

Unfortunately, ORM frameworks inevitably bring concerns to the performance and scalability of web applications, whose multi-stack nature demands cross-stack performance understanding and optimization. On one hand, it is difficult for application compilers or developers to optimize the interaction between the application and the underlying DBMS, as they are unaware of how their code would translate to queries by the ORM. On the other hand, ORM framework and the underlying DBMS are unaware of the high-level application semantics and hence cannot generate efficient plans to execute queries. Making things even worse, data-related performance and scalability problems are particularly difficult to expose during in-house testing, as they might only occur with large amounts of data that only arises after the application is deployed.

To tackle these problems, on the performance side, I have conducted a comprehensive study of performance issues in real-world DB-backed web applications [102] and developed techniques to tackle each main category of performance problems revealed by my study [101, 103], detecting and fixing *thousands* of performance issues in latest versions of popular web applications along the way. On the correctness side, I have conducted an in-depth empirical study about data constraint problems in real-world DB-backed web applications and developed static checkers that identified *thousands* of data constraint issues [98], which can lead to software crashes and severe failures.

CHAPTER 2
A COMPREHENSIVE STUDY OF PERFORMANCE ISSUES IN DATABASE-BACKED WEB APPLICATIONS

2.1 Overview

In this chapter, we present a comprehensive two-pronged empirical study on a set of 12 Rails applications representing 6 common categories that exercise a wide range of functionalities provided by the ORM framework and DBMS. The study aims to answer three research questions:

- **RQ 1**: How well do real-world database-backed web applications perform as the amount of application data increases?
- **RQ 2**: What are the common root causes of performance and scalability issues in such applications?
- **RQ 3**: What are the potential solutions to such issues and can they be applied automatically?

We choose Rails as it is a popular ORM framework [19]. We carefully examine 140 fixed performance issues randomly sampled from the bug-tracking systems of these 12 applications. This helps us understand how well these applications performed on real-world data *in the past*, and what types of performance and scalability problems have been discovered and fixed by end-users and developers in *past versions*.

To complement the above study, we also conduct thorough profiling and code review of the latest versions of these 12 applications. This investigation helps us understand how these applications *currently* perform on our carefully synthesized data (to be explained in Section 2.2), what types of performance and scalability problems still exist in the *latest versions*, and how they can be fixed.

In terms of findings, for **RQ1**, our profiling in Section 2.3 shows that, under workload that is no larger than today's typical workload, 11 out of 12 studied applications contain pages in their latest versions that take more than two seconds to load and also pages that scale super-linearly. Compared to client-side computation (e.g., executing JavaScript functions in the browser), server-side computation takes more time in most time-consuming pages and often scales much worse. These results motivate research to tackle server-side performance problems in ORM applications.

For **RQ2**, we generalize 9 ORM performance anti-patterns by thoroughly studying about 200 real-world performance issues, with 140 collected from 12 bug-tracking systems and 64 manually identified by us based on profiling the same set of ORM applications (Section 5.4). We group these 9 patterns into three major categories—ORM API misuses, database design problems, and trade-offs in application design. All but one of these patterns exist both in previous versions (i.e., fixed and recorded in bug-tracking systems) and the latest versions (i.e., discovered by us through profiling and code review) of these applications. 6 of these anti-patterns appear profusely in more than half of the studied applications. While a few of them have been identified in prior work, the majority of these anti-patterns have *not* been researched in the past.

Finally, for **RQ3**, we manually design and apply fixes to the 64 performance issues in the latest versions of these 12 ORM applications (Section 4.4). Our fixes achieve 2$\times$ median speedup (and up to 39 $\times$) in server-side performance improvement, and reduce the average end-to-end latency of 39 problematic web pages in 12 applications from 4.17 seconds to 0.69 seconds, making them interactive. Most of these optimizations follow generic patterns that we believe can be automated in the future through static analysis and code transformations. As a proof of concept, we implement a simple static checker based on our findings and use it to find hundreds of API misuse performance problems in the latest versions of these applications (Section 2.6).

Table 2.1: Details of the applications chosen in our study

Category	Abbr.	Name	Stars	Commits	Contributors
Forum	Ds	Discourse	21238	22501	568
	Lo	Lobster	1304	1000	48
Collaboration	Gi	Gitlab	19255	49810	1276
	Re	Redmine	2399	13238	6
E-commerce	Sp	Spree	8331	17197	709
	Ro	Ror_ecommerce	1109	1727	21
Task-management	Fu	Fulcrum	1502	697	44
	Tr	Tracks	835	3512	62
Social Network	Da	Diaspora	11183	18734	335
	On	Onebody	1592	1220	6
Map	OS	Openstreetmap	664	8000	112
	FF	Fallingfruit	41	1106	7

2.2 Profiling & Study Methodology

This section explains how we profile ORM applications and study their bug-tracking systems, with the goal to understand how they perform and scale in both their latest and previous versions.

2.2.1 Application Selection

As mentioned in Section 2.1, we focus on Rails applications. Since it is impractical to study all open-source Rails applications (about 200 thousand of them on GitHub [20]), we focus on 6 popular application categories[1] as shown in Table 2.1. These 6 categories cover 90% of all Rails applications with more than 100 stars on GitHub. They also cover a variety of database-usage characteristics, such as transaction-heavy (e.g., e-commerce), read-intensive (e.g., social networking), and write-intensive (e.g., forums). Furthermore, they cover both

1. We use the category information as provided by the application developers. For example, Diaspora is explicitly labeled 'social-network' [11].

graphical interfaces (e.g., maps) and traditional text-based interfaces (e.g., forums).

We study the top 2 most popular applications in each category, based on the number of "stars" on GitHub. These 12 applications shown in Table 2.1 have been developed for 5 to 12 years. They are all in active use and range from 7K lines of code (Lobsters [29]) to 145K lines of code (Gitlab [21]).

2.2.2 Profiling Methodology

Populating databases To profile an ORM application, we need to populate its database. Without access to the database contents in the deployed applications, we collect real-world statistics of each application based on its public website (e.g., `https://gitlab.com/explore` for Gitlab [21]) or similar application's website (e.g., amazon [2] statistics for e-commerce type applications). We then synthesize database contents based on these statistics along with application-specific constraints. Specifically, we implement a crawler that fills out forms on the application webpages hosted on our profiling servers with data automatically generated based on the data type constraints. Our crawler carefully controls how many times each form is filled following the real-world statistics collected above.

Take Gitlab as an example, an application that allows user to manage projects and git repositories. We run a crawler on our own Gitlab installation. Under each generated user account, the crawler first randomly decides how many projects this user should own based on the real-world statistics collected from `https://gitlab.com/explore` shown in Table 2.2, say N, and then fills the create project page N times. The crawler continues to create new project commits, branches, tags, and others artifacts in this manner.

Other applications are populated similarly, and we skip the details due to space constraints. Virtual-machine images that contain all these applications and our synthetic database content, as well as data-populating scripts, are available at our project website [25].

Table 2.2: Some of Gitlab statistics for workload synthesis

#projects	#users	#commits	#projects	#branches	#projects
≤ 1	74678	≤ 1	115246	≤ 1	224551
1 - 5	31009	1 - 5	51499	1 - 5	54171
5 -10	5063	5 -10	26429	5 -10	7097
10 - 20	2133	10 - 20	25797	10 - 20	4429
20 - 100	1116	20 - 100	41939	20 - 100	3996
100 - 1000	97	100 - 1000	23407	100 - 1000	3644
> 1000	4	> 1000	14098	> 1000	527

Statistics about (1) the number of users who own certain number of projects; and the number of projects that have certain number of (2) commits and (3) branches.

Table 2.3: Database sizes in MB

#records	Ds	Lo	Gi	Re	Sp	Ro	Fu	Tr	Da	On	FF	OS
200	10	10	11	11	46	30	3	3	10	17	12	9
2000	25	100	135	35	83	98	10	16	39	53	14	14
20000	182	982	764	224	340	233	68	62	200	259	32	62

Scaling input data We synthesize three sets of database contents for each application that contain 200, 2000, and 20,000 records in its main database table, which is the one used in rendering homepage, such as the `projects` table for Gitlab and Redmine, the `posts` table for social network applications, etc. Other tables' sizes scale up accordingly following the data distribution statistics discussed above. The total database sizes (in MB) under these three settings are shown in Table 4.3.

When we discuss an application's *scalability*, we compare its performance among the above three settings. When we discuss an application's *performance*, we focus on the 20,000-record setting, which is a *realistic* setting for all the applications under study. In fact, based on the statistics we collect, the number of main table records of every application under study is usually larger than 20,000 in public deployments of the applications. For example, Lobsters and Tracks' main tables hold the fewest records, 25,000 and 26,000, respectively. Many applications contain more than 1 million records in their main tables — Spree's official website contains almost 500 million products, Fallingfruit's official website contains more than 1 million locations on map, etc.

Identifying problematic actions Next, we profile applications to identify actions with potential performance problems. We deploy an application's latest version under Rails production mode on AWS m4.xlarge instance [6] with populated databases. We run a Chrome-based crawler [7] on another AWS instance to visit links repeatedly and randomly for 2 hours to collect performance profiles for every action in an application.[2] We repeat this for all three sets of databases shown in Table 4.3, and each set is repeated for 3 times. We then process the log produced by both Chrome and the Rails Active Support Instrumentation API [1] to obtain the average end-to-end loading time for every page, the detailed performance breakdown, as well as issued database queries.

For each application, we firstly identify the top 10 most time-consuming controller actions, among which we further classify an action A as *problematic* if it either spends more than one second on the server side, meaning that the corresponding end-to-end loading time would likely approach two seconds, making it undesirable for most users [49]; or its performance grows super-linearly as the database size increases from 200 to $2,000$ and then to $20,000$ records. The identified actions are the target of our study on performance and scalability problems as we describe in Section 5.4 and 4.4.

2.2.3 Report-Study Methodology

To complement the above profiling that examines the latest version of an application using our synthetic datasets, we also study the performance issues reported by users based on real-world workloads and fixed by developers for past versions of these ORM applications, so that we can understand how well these deployed applications performed in the past.

To do so, we examine each application's bug-tracking system. For 6 applications that contain fewer than 1000 bug reports, as shown in Table **??**, we manually check every bug report. For applications with 1000 to 5000 bug reports, we randomly sample 100 bug reports

2. The database size will increase a little bit during profiling as some pages contain forms, but the overall increase is negligible and does not affect our scalability comparison.

Table 2.4: Numbers of sampled and total issue reports

Ds	Lo	Gi	Re	Sp	Ro	Fu	Tr	Da	On	FF	OS
17	7	22	22	28	2	2	12	13	5	3	7
4607	220	18038	12117	4805	114	158	1470	3206	400	17	650

The upper row shows the number of reports sampled for our study; the lower row shows the total number of reports in each application's bug-tracking system.

that have been fixed and contain the keywords *performance*, *slow*, or *optimization*. For Redmine and Gitlab, which have more than 10,000 bug reports, we sample 200 from them in the same way. By manually checking each report's discussion, source code, and patches, we identify the ones that truly reflect performance problems related to data processing on the server side. Every bug report is cross-checked by at least two authors. This results in 140 reports in total from all 12 applications, as shown in Table **??**.

2.2.4 Threats to Validity

Threats to the validity of our study could come from multiple sources. Applications beyond these 12 applications may not share the same problems as these 12 applications. The profiling workload synthesized by us may not accurately represent the real-world workload. The machine and network settings of our profiling may be different from real users' setting. Our study of each application's bug-tracking system does not consider bug reports that are not fixed or not clearly explained. Despite these aspects, we have made our best effort in conducting a comprehensive and unbiased study, and we believe our results are general enough to guide future research on improving performance of ORM applications.

2.3 Profiling Results

End-to-end loading time We identify the 10 pages with the most loading time for every application under the 20,000-record database configuration and plot their average end-to-end

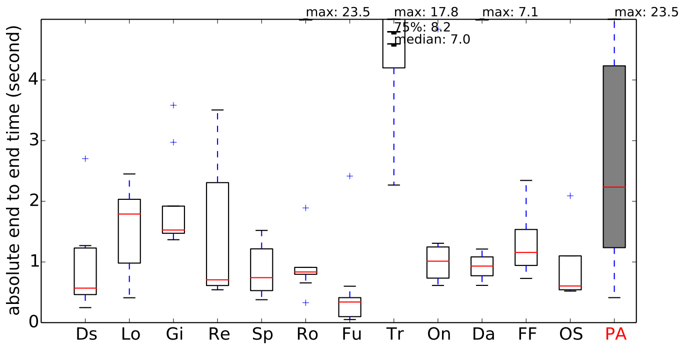


Figure 2.1: End-to-end page loading time

Measured for top 10 time-consuming pages per application. Box: 25 to 75 percentile; Red line: median; PA: problematic actions from all 12 applications (see Section 2.2.2).

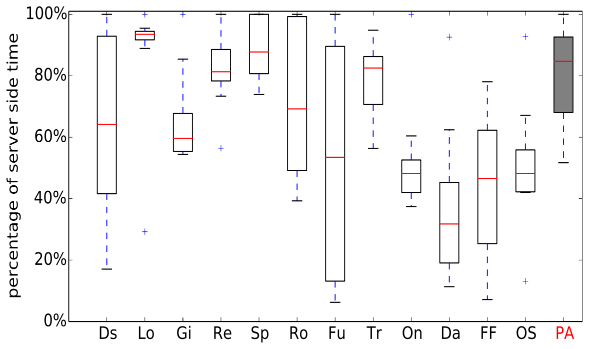


Figure 2.2: Percentage of server time among end-to-end time

Measured for top 10 most time-consuming pages per application. Red line: median; PA: problematic actions from all 12 applications (see Section 2.2.2) page loading time in Figure 2.1. 11 out of 12 applications have pages whose average end-to-end loading time (i.e., from browser sending the URL request to page finishing loading) exceeds 2 seconds; 6 out of 12 applications have pages that take more than 3 seconds to load. *Tracks* performs the worst: all of its top 10 most time-consuming pages take more than 2 seconds to load. Note that, our workload is *smaller* or, for some applications, *much smaller* than today's real-world workload. Considering how the real-world workload's size will continue growing, these results indicate that performance problems are prevalent and critical for deployed Rails applications.

Server vs. client We break down the end-to-end loading time of the top 10 pages in each application into server time (i.e., time for executing controller action, including view rendering and data access, on Rails server), client time (i.e., time for loading the DOM in the browser), and network time (i.e., time for data transfer between server and browser). As shown in Figure 2.2, server time contributes to at least 40% of the end-to-end-latency for more than half of the top 10 pages in all but 1 application.[3] Furthermore, over 50% of problematic pages spend more than 80% of the loading time on Rails server, as shown by the rightmost bar (labeled **PA**) in Figure 2.2. This result further motivates us to study the performance problems on the server side of ORM applications.

Problematic server actions Table 2.5 shows the number of problematic actions for each application identified using the methodology discussed in Section 2.2.2. In total, there are 40 problematic actions identified from the top 10 most time-consuming actions of every application. Among them, 34 have scalability problems and 28 take more than 1 second of server time. Half of the pages that correspond to these 40 problematic actions take more than 2 seconds to load, as shown in the rightmost bar (labeled **PA**) in Figure 2.1. In addition, we find 64 performance issues in these 40 problematic actions, and we will discuss them in

3. Part of the server time could overlap with the client time or the network time. However, our measurement shows that the overlap is negligible.

Table 2.5: Number of problematic actions in each application

App	Ds	Lo	Gi	Re	Sp	Ro	Fu	Tr	Da	On	FF	OS
slow	0	0	1	1	3	0	0	0	0	1	0	0
not-scalable	1	1	0	0	0	0	2	0	1	2	3	1
slow & not-scalable	0	5	1	2	0	2	1	10	1	0	1	0

detail in Section 5.4.

2.4 Causes of Inefficiencies

After studying the 64 performance issues in the 40 problematic actions and the 140 issues reported in the applications' bug-tracking systems, we categorize the inefficiency causes into three categories: ORM API misuses, database design, and application design. In the following we discuss these causes and how developers have addressed them. We believe these causes apply to applications built using other ORM frameworks as well, as we will discuss in Section 2.7.

2.4.1 ORM API Misuses

About half of the performance issues that we studied suffer from API misuses. In these cases, performance can be improved by changing how the Rails APIs are used without modifying program semantics or database design. While some of these misuses appear simple, making the correct decision requires deep expertise in the implementation of the ORM APIs and query processing.

Inefficient Computation (IC)

In these cases, the poorly performing code conducts useful computation but inefficiently. Such cases comprise more than 10% of the performance issues in both bug reports and problematic actions.

Table 2.6: Inefficiency causes across 12 applications

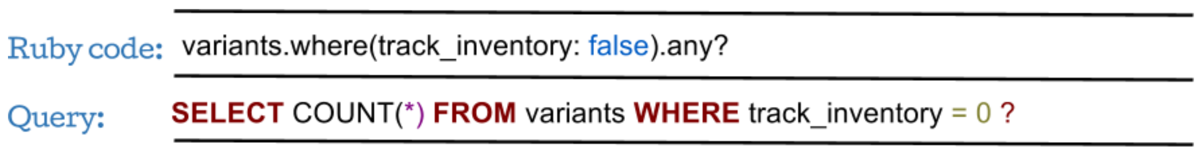
	Ds	Lo	Gi	Re	Sp	Ro	Fu	Tr	Da	On	FF	OS	Sum
ORM API Misuse													
IC	0	0	0	0	0	1	0	1	2	2	2	0	8
	0	0	3	6	5	0	0	2	2	0	0	0	18
UC	0	3	0	0	0	0	0	0	0	0	2	0	5
	1	0	3	4	4	1	0	1	2	1	0	0	17
ID	0	1	0	0	3	2	0	3	2	3	0	1	15
	3	1	4	5	11	0	0	2	1	2	0	0	29
UD	0	0	1	0	0	0	0	0	0	0	0	0	1
	2	0	3	1	2	0	0	0	0	0	0	0	8
IR	0	3	1	0	0	0	0	1	0	0	0	0	5
Database Design Problems													
MF	0	0	0	1	0	0	0	0	0	0	1	1	3
	0	2	0	0	2	0	0	0	0	0	1	0	5
MI	0	1	0	0	0	0	0	0	0	0	2	0	3
	3	1	4	6	3	0	0	3	5	1	1	3	30
Application Design Tradeoffs													
DT	1	0	0	2	0	2	6	10	0	1	0	0	22
	5	1	1	0	0	1	0	3	1	0	0	2	14
FT	0	2	0	0	0	0	0	0	0	0	0	0	2
	3	2	4	0	1	0	2	1	2	1	1	2	19
Sum	18	17	24	25	31	7	8	27	17	11	10	9	204

Data with white background shows 64 issues from 40 problematic actions
Data with gray background shows 140 issues from 12 bug-tracking systems

IC:	Inefficient Computation	MF:	Missing Fields
UC:	Unnecessary Computation	MI:	Missing Indexes
ID:	Inefficient Data Accessing	DT:	Content Display Trade-offs
UD:	Unnecessary Data Retrieval	FT:	Functionality Trade-offs
IR:	Inefficient Rendering		

60mm

(a) t

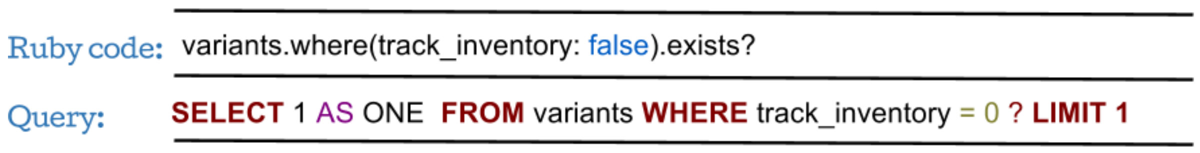


```
Ruby code: variants.where(track_inventory: false).any?
Query:     SELECT COUNT(*) FROM variants WHERE track_inventory = 0 ?
```

Figure 2.3: Inefficient

60mm

(a) t



```
Ruby code: variants.where(track_inventory: false).exists?
Query:     SELECT 1 AS ONE  FROM variants WHERE track_inventory = 0 ? LIMIT 1
```

Figure 2.4: Efficient

Figure 2.5: Different APIs cause huge performance difference

Inefficient queries The same operation on persistent data can be implemented via different ORM calls. However, the performance of the generated queries can be drastically different. This problem has not been well studied before for ORM applications.

Figure 2.5 shows two ways that an online shopping system checks if there are product `variants` whose inventory are not tracked. The Ruby code differs only in the use of `any?` vs `exists?`. However, the performance of the generated queries differs substantially: the generated query in Figure 3.2 scans all records in the `variants` table to compute the count if no index exists, but that in Figure 2.4 only needs to scan and locate the first `variant` record where the predicate evaluates to true. `Spree` developers discovered and fixed this problem in `Spree-6720`.[4] Our profiling finds similar problems. For example, simply replacing `any?` with `exists?` in a problematic action of `OneBody` improves server time by 1.7×. Our static checker that will be discussed in Section 2.7 finds that this is a common problem as it appears in the latest versions of 9 out of 12 applications under study.

Another common problem is developers using API calls that generate queries with unnecessary ordering of the results. For example, Ror, Diaspora, and Spree developers use

4. We use `A-n` to denote report number `n` in application `A`'s bug-tracking system.

`Object.where(c).first` to get an object satisfying predicate `c` instead of `Object.find_by(c)`, not realizing that the former API orders `Objects` by primary key after evaluating predicate `c`. As a fix, both Gitlab and Tracks developers explicitly add `except(:order)` in the patches to eliminate unnecessary ordering in the queries, further showing how simple changes can lead to drastic performance difference.

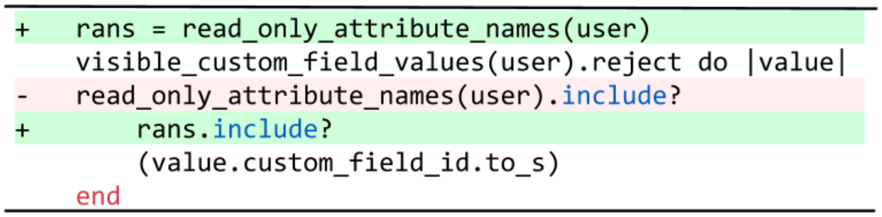
Moving computation to the DBMS As the ORM framework hides the details of query generation, developers often write code that results in multiple queries being generated. Doing so incurs extra network round-trips, or running computation on the server rather than the DBMS, which leads to performance inefficiencies.

For example, the patch of `Spree-6720` replaces `if(exist?) find; else create` with `find_or_create_by`, where the latter combines two queries that are issued by `exist` and `find/create` into one. The patch of `Spree-6950` replaces `pluck(:total).sum` with `sum(:total)`. The former uses `pluck` to issue a query to load the `total` column of all corresponding records and then computes the sum in memory, while the latter uses `sum` to issue a query that directly performs the sum in the DBMS without returning actual records to the server. The patch of `Gitlab-3325` replaces `pluck(:id)+pluck(:id)`, which replaces two queries and an in-memory union via `+` with one SQL `UNION` query, in effect moving the computation to the DBMS. Such API misuses are very common and occur in many applications as we will discuss in Section 2.7.

There are also more complicated cases where a loop implemented in Ruby can be completely pushed down to DBMS, which has been addressed in previous work using program synthesis [58].

Moving computation to the server Interestingly, there are cases where the computation should be moved to the server from the DBMS. As far as we know, this issue has not been studied before.

For example, in the patch of `Spree-6819`, developers replace `Objects.count` with `Objects.size`



```
+    rans = read_only_attribute_names(user)
     visible_custom_field_values(user).reject do |value|
-    read_only_attribute_names(user).include?
+        rans.include?
         (value.custom_field_id.to_s)
     end
```

Figure 2.6: A loop-invariant query in Redmine

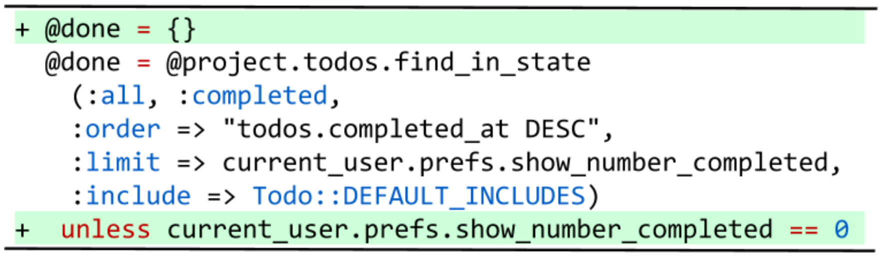
in 17 different locations, as `count` always issues a `COUNT` query while `size` counts the `Objects` in memory if they have already been retrieved from the database by earlier computation. Such issues are also reported in `Gitlab-17960`.

Summary Rails, like other ORM frameworks, lets developers implement a given functionality in various ways. Unfortunately, developers often struggle at picking the most efficient option. The deceptive names of many Rails APIs like `count` and `size` make this even more challenging. Yet, we believe many cases can be fixed using simple static analyzers, as we will discuss in Section 2.7.

Unnecessary Computation (UC)

More than 10% of the performance issues are caused by (mis)using ORM APIs that lead to unnecessary queries being issued. This type of problems has not been studied before.

Loop-invariant queries Sometimes, queries are repeatedly issued to load the *same* database contents and hence are unnecessary. For instance, Figure 2.6 shows the patch from `redmine-23334`. This code iterates through every custom field `value` and retains only those that `user` has write access to. To conduct this access-permission checking, in every iteration, `read_only_attribute_names(user)` issues a query to get the names of all read-only fields of `user`, as shown by the red highlighted line in the figure. Then, if `value` belongs to this read-only set, it will be excluded from the return set of this function (i.e., the `reject` at the beginning of the loop takes effect). Here, the `read_only_attribute_names(user)` query returns exactly the same result



```
+ @done = {}
  @done = @project.todos.find_in_state
    (:all, :completed,
    :order => "todos.completed_at DESC",
    :limit => current_user.prefs.show_number_completed,
    :include => Todo::DEFAULT_INCLUDES)
+   unless current_user.prefs.show_number_completed == 0
```

Figure 2.7: A query with known results in Tracks

during every iteration of the loop and causes unnecessary slowdowns. As shown by the green lines in figure, Redmine developers hoist loop invariant `read_only_attribute_names(user)` outside the loop and achieve more than 20× speedup for the corresponding function for their workload. Similar issues also occur in Spree and Discourse.

Dead-store queries In such cases, queries are repeatedly issued to load *different* database contents into the same memory object while the object has not been used between the reloads. For example, in Spree, every shopping transaction has a corresponding `order` record in the `orders` table. This table has a `has_many` association relationship with the `line_items` table, meaning that every order contains multiple lines of items. Whenever the user updates his/her shopping cart, the `line_items` table would change, at which point the old version of Spree always uses an `order.reload` to make sure that the in-memory copy of `order` and its associated `line_items` are up-to-date. Later on, developers realize that this repeated reload is unnecessary, because the content of `order` is not used by the program until check out. Consequently, in `Spree-6379`, developers remove many `order.reload` from model classes, and instead add it in a few places in the `before_payment` action of the `checkout` controller, where the `order` object is to be used.

Queries with known results A number of issues are due to issuing queries whose results are already known, hence incurring unnecessary network round trips and query processing time. An example is in `Tracks-63`. As shown in Figure 2.7, the code originally issues a query to retrieve up to `show_number_completed` number of completed tasks. Clearly, when `show_number_completed` is 0, the query always returns an empty set due to `limit` being 0. Developers later realize that 0 is a very common setting for `show_number_completed`. Consequently, they applied the patch shown in Figure 2.7 to only issue the query when needed.

Summary While similar issues in general purpose programs can be eliminated using classic compiler optimization techniques (e.g., loop invariant motion, dead-store elimination), doing so for ORM applications is difficult as it involves understanding database queries. We are unaware of any compilers that perform such transformations.

Inefficient Data Accessing (ID)

Problems under this category suffer from data transfer slow downs, including not batching data transfers (e.g., the well-known "N+1" problem) or batching too much data into one transfer.

Inefficient lazy loading As discussed in Section **??**, when a set of objects O in table T_1 are requested, objects stored in table T_2 associated with T_1 and O can be loaded together through eager loading. If lazy loading is chosen instead, one query will be issued to load N objects from T_1, and then N separate queries have to be issued to load associations of each such object from T_2. This is known as the "N+1" query problem. While prior work has studied this problem [30, 57, 8], we find it still prevalent: it appears in 15 problematic actions and 9 performance issues in our study.

Figure 2.8 shows an example that we find in the latest version of Lobsters, where the deleted code retrieves 50 `mods` objects. Then, for each `mod`, a query is issued to retrieve its associated `story`. Using eager loading in the added line, all 51 queries (and hence 51 network round-trips) will be combined together. In our experiments, the optimization reduces the end-to-end loading time of the corresponding page from 1.10 seconds to 0.34 seconds.

```
-   mods = Moderation.limit(50)
+   mods = Moderation.includes(:story).limit(50)
    mods.each do |mod|   render mod.story   end
```

Figure 2.8: Inefficient lazy loading in Lobsters

```
   products.includes([{ :variants => [:images,
-  { :option_values => :option_type }
   ], :master => [:images, :default_price]}])
```

Figure 2.9: Inefficient eager loading in Spree

Inefficient eager loading However, always loading data eagerly can also cause problems. Specifically, when the associated objects are too large, loading them all at once will create huge memory pressure and even make the application unresponsive. In contrast to the "N+1" lazy loading problem, there is little support for developers to detect eager loading problems.

In `Spree-5063`, a Spree user complains that their installation performs very poorly on the product search page. Developers found that the problem was due to eager loading shown in Figure 2.9. In the user's workload, while loading 405 `products` to display on the page, eager loading causes 13811 related `variants` products containing 276220 `option_values` (i.e., product information data) to be loaded altogether, making the page freeze. As shown in Figure 2.9, the patch delays the loading of `option_values` fields of `variants` products. Note that these `option_values` are needed by later computation, and the patch delays but not eliminates their loading.

Inefficient updating Like the "N+1" problem, developers would issue N queries to update N records separately (e.g., `objects.each |o| o.update end`) rather than merging them into one update (e.g., `objects.update_all`). This is reported in Redmine and Spree, and our static checker (to be discussed in Section 2.7) finds this to be common in the latest versions of 6 out of the 12 studied applications.

Unnecessary Data Retrieval (UD)

Unnecessary data retrieval happens when software retrieves persistent data that is not used later. Prior work has identified this problem in applications built using both Hibernate [56] and Rails [95]. In our study, we find this continues to be a problem in one problematic action in the latest version of Gitlab and 9 performance issue reports. Particularly, fixing the unnecessary data retrieval in the latest version of Gitlab can drop the end-to-end loading time of its `Dashboard/Milestones/index` page from 3.0 to 1.1 seconds in our experiments. We also see some unnecessary data retrieval caused by simple misuses of APIs that have similar names — `map(&:id)` retrieves the whole record and then returns the `id` field, yet `pluck(:id)` only retrieves the `id` field.

Inefficient Rendering (IR)

IR reflects a trade-off between readability and performance when a view file renders a set of objects. It has not been studied before.

Given a list of objects to render, developers often use a library function, like `link_to` on Line 4 of Figure 2.10(a), to render one object and encapsulate it in a partial view file such as `_milestone.html.haml` in Figure 2.10(a). Then, the main view file `index.html.haml` simply applies the partial view file repeatedly to render all objects. The inefficiency is that a rendering function like `link_to` is repeatedly invoked to generate very similar HTML code. Instead, the view file could generate the HTML code for one object, and then use simple string substitution, such as `gsub` in Figure 2.10(b), to quickly generate the HTML code for the remaining objects, avoiding redundant computation. The latter way of rendering degrades code readability, but improves performance substantially when there are many objects to render or with complex rendering functions.

Although slow rendering is complained, such transformation has not yet been proposed by issue reports. Our profiling finds such optimization speeds up 5 problematic actions by

```
   milestones/index.html.haml
milestones.each do |milestone|
   render milestone
end
   milestones/_milestone.html.haml
link_to milestone.title, milestone.description
```

(a) Inefficient partial rendering

```
   milestones/index.html.haml
a = link_to `m_title`, `m_descr`
milestones.each do |milestone|
    a.gsub(`m_title`, milestone.title)
     .gsub(`m_descr`, milestone.description)
end
```

(b) Efficient partial rendering

Figure 2.10: Inefficient partial rendering in Gitlab

2.5× on average.

2.4.2 Database Design Problems

Another important cause of performance problems is suboptimal database design. Fixing it requires changing the database schema.

Missing Fields (MF)

Deciding which object field to be physically stored in database is a non-trivial part of database schema design. If a field can be easily derived from other fields, storing it in database may waste storage space and I/O time when loading an object; if it is expensive to compute, not storing it in database may incur much computation cost. Deciding when a property should be stored persistently is a general problem that has not been studied in prior work.

For example, when we profile the latest version of Openstreetmap [31], a collaborative editable map system, we find that a lot of time is spent on generating a `location_name` string for every diary based on the diary's longitude, latitude, and language properties stored in the `diary_entry` table. Such slow computation results in a problematic action taking 1 second to show only 20 diaries. However, the `location_name` is usually a short string and remains the same value since the location information for a diary changes infrequently. Storing this string physically as a database column avoids the expensive computation. We evaluate this optimization and find it reducing the action time to only 0.36 second.

We observe similar problems in the bug reports of Lobster, Spree, and Fallingfruit, and in the latest version of Redmine, Fallingfruit, and Openstreetmap. Clearly, developers need help on performance estimation to determine which fields to persistently store in database tables.

Missing Database Indexes (MI)

Having the appropriate indexes on tables is important for query processing and is a well-studied problem [86]. As shown in Table 2.6, missing index is the most common performance problem reported in ORM application's bug tracking systems. However, it only appears in three out of the 40 problematic actions in latest versions. We speculate that ORM developers often do not have the expertise to pick the optimal indexes at the design phase and hence add table indexes in an incremental way depending on which query performance becomes a problem after deployment.

2.4.3 Application Design Trade-offs

Developers fix 33 out of the 140 issue reports by adjusting application display or removing costly functionalities. We find similar design problems in latest versions of 7 out of 12 ORM applications. It is impractical to completely automate display and functionality design. However, our study shows that ORM developers need tool support, which does not exist yet, to be more informed about the performance implication of their application design decisions.

Content Display Trade-offs (DT)

In our study, the most common cause for scalability problems is that a controller action displays *all* database records satisfying certain condition in one page. When the database size increases, the corresponding page takes a lot of time to load due to the increasing amount of data to retrieve and render. This problem contributes to 15 out of the 34 problematic actions that do not scale well in our study. It also appears in 7 out of 140 issue reports, and is *always* fixed by pagination, i.e., display only a fixed number of records in one page and allow users to navigate to remaining records.

For example, in `Diaspora-5335` developers used the `will_paginate` library [32] to render 25 contacts per page and allow users to see the remaining contacts by clicking the navigation bar at the bottom of the page, instead of showing all contacts within one page as in the old version. Clearly, good UI designs can both enhance user experience and improve application performance.

Unfortunately, the lack of pagination still widely exists in latest versions of ORM applications in our study. This indicates that ORM developers need database-aware performance-estimation support to remind them of the need to use pagination in webpage design.

Application Functionality Trade-offs (FT)

It is often difficult for ORM developers to estimate performance of a new application feature given that they need to know what queries will be issued by the ORM, how long these queries will execute, and how much data will be returned from the database. In our study, all but two applications have performance issues fixed by developers through removing functionality.

For example, `Tracks-870` made a trade-off between performance and functionality by removing a sidebar on the resulting page. This side bar retrieves and displays all the projects and contexts of the current user, and costs a lot of time for users who have participated in many projects. In the side-bar code, the only data-related part is simply a

`@sidebar.active_projects` expression, which seems like a trivial heap access but actually issues a `SELECT` query and retrieves a lot of data from the database.

As another example, our profiling finds that the `story.edit` action in the latest version of Lobsters takes 1.5 seconds just to execute one query that determines whether to show the `guidelines` for users when they edit stories, while the entire page takes 2 seconds to load altogether. Since the `guidelines` object only takes very small amount of space to show on the resulting page, removing such checking has negligible impact to the application functionality, yet it would speed up the loading time of that page a lot.

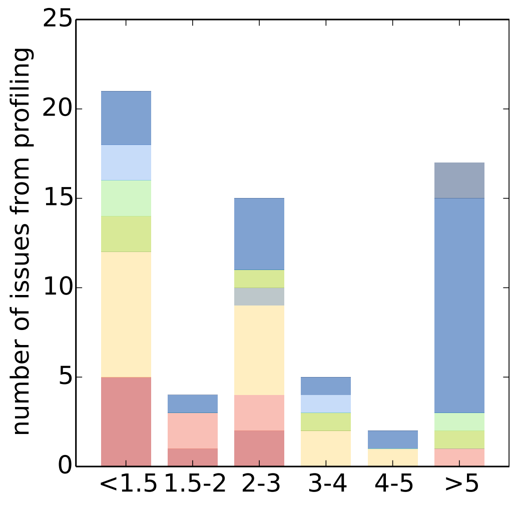
In general, performance estimation for applications built using ORMs is important yet has not been done before. It is more difficult as compared to traditional applications due to multiple layers of abstraction. We believe combining static analysis with query scalability estimation [51, 63] will help developers estimate application performance, as we will discuss in Section 2.7.

2.5 Fixing the Inefficiencies

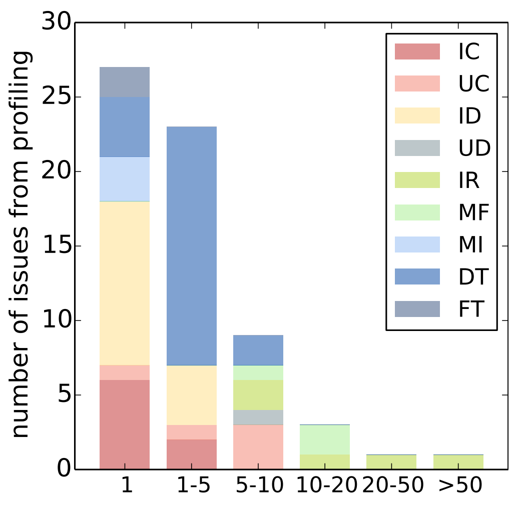
After identifying the performance inefficiencies in the 40 problematic actions across the 12 studied applications, we manually fix each of them and measure how much our fixes improve the performance of the corresponding application webpages. Our goal is to quantify the importance of the anti-patterns discussed in Section 5.4.

2.5.1 Methodology

We use the same 20,000-record database configuration used in profiling to measure performance improvement. For a problematic action that contains multiple inefficiency problems, we fix one at a time and report the speedup for each individual fix. To fix API-use problems, we change model/view/control files that are related to the problematic API uses; to add missing indexes or fields, we change corresponding Rails migration files; to apply pagination,



(a) Server-time speedup (×)



(b) Line of code changes

Figure 2.11: Performance fixes and LOC involved

we use the standard `will_paginate` library [32]. We carefully apply fixes to make sure we do not change the program semantics. Finally, for two actions in Lobster, we eliminate the expensive checking about whether to show user guidelines, as discussed in Section 2.4.3.

2.5.2 Results

In total, 64 fixes are applied across 39 problematic actions [5] to solve the 64 problems listed in Table 2.6.

Speedup of the fixes Figure 2.11(a) shows the amount of server-time speedup and the sources of the speedup broken down into different anti-patterns as discussed in Section 5.4.

Many fixes are very effective. About a quarter of them achieve more than 5× speedup, and more than 60% of them achieve more than 2× speedup. Every type of fixes has at least one case where it achieves more than 2× speedup. The largest speed-up is around 39 × achieved by removing unnecessary feature in `StoriesController.new` action in Lobsters, i.e., the example we discussed in Section 2.4.3.

5. Among the 40 problematic actions identified by our profiling, 1 of them (from GitLab) spends most of its time in file-system operations and cannot be sped up unless its core functionality is modified.

There are 40 fixes that alter neither the display nor the functionality of the original application. That is, they fix the anti-patterns discussed in Section 2.4.1 and 2.4.2. They achieve an average speedup of 2.2×, with a maximum of 9.2× speedup by adding missing fields in `GanttsController.show` from Redmine.

For all 39 problematic actions, many of which benefit from more than one fix, their average server time is reduced from 3.57 seconds to 0.49 seconds, and the corresponding end-to-end page loading time is reduced from 4.17 seconds to 0.69 seconds, including client rendering and network communication. In other words, by writing code that contains the anti-patterns discussed earlier, developers degrade the performance of their applications by about 6×.

We have reported these 64 fixes to corresponding developers. So far, we have received developers' feedback for 14 of them, all of which have been confirmed to be true performance problems and 7 have already been fixed based on our report.

Simplicity of the fixes Figure 2.11(b) shows the lines of code changes required to implement the fixes. The biggest change takes 56 lines of code to fix (for an inefficient rendering (IR) anti-pattern), while the smallest change requires only 1 line of code in 27 fixes. More than 78% of fixes require fewer than 5 lines. In addition, among the fixes that improve performance by 3× or more, more than 90% of them take fewer than 10 lines of code. Around 60% of fixes are intra-procedural, involving only one function.

These results quantitatively show that there is still a huge amount of inefficiency in real-world ORM applications. Much inefficiency can be removed through few lines of code changes. A lot of the fixes can potentially be automated, as we will discuss in Section 2.7.

Table 2.7: API misuses we found in the latest versions

App.	①	②	③	④	⑤	⑥	⑦	⑧	⑨	SUM
Ds	8	61	0	0	6	6	3	0	1	85
Lo	1	38	0	0	0	5	1	0	0	45
Gi	7	3	0	1	6	3	3	0	0	23
Re	3	32	0	1	16	7	0	0	0	59
Sp	2	10	0	0	0	0	7	1	0	20
Ro	0	7	0	1	1	0	2	0	0	11
Fu	0	0	0	0	2	0	0	0	0	2
Tr	4	22	0	1	3	0	0	0	0	30
Da	5	42	1	1	0	8	0	0	0	57
On	10	60	0	0	6	0	0	0	0	76
FF	2	0	0	2	0	0	0	0	0	4
OS	0	12	0	0	2	2	0	0	0	16
SUM	42	287	1	7	42	31	16	1	1	428

①: `any?` ⇒ `exists?` ②: `where.first` ⇒ `find_by`
③: `*` ⇒ `*.except(:order)` ④: `each.update` ⇒ `update_all`
⑤: `.count` ⇒ `.size` ⑥: `.map` ⇒ `.pluck`
⑦: `pluck.sum` ⇒ `sum` ⑧: `pluck + pluck` ⇒ `SQL-UNION`
⑨: `if exists?  find else create end` ⇒ find_or_create_by

2.6 Finding more API misuses

Some problems described in Section 2.4.1 are about simple API misuses. We identify 9 such simple misuse patterns, as listed in Table 2.7, and implement a static analyzer to search for their existence in latest versions of the 12 ORM applications. Due to space constraints, we skip the implementation details. To recap, these 9 API patterns cause performance losses due to "An Inefficient Query" (①, ②, ③), "Moving Computation to the DBMS" (⑦, ⑧, ⑨), "Moving Computation to the Server" (⑤), "Inefficient Updating" (④), and "Unnecessary Data Retrieval" (⑥), as discussed in Section 2.4.1.

As shown in Table 2.7, every API misuse pattern still exists in at least one application's latest version. Worse, 4 patterns each occur in over 30 places across more than 5 applications.

We have checked all these 428 places and confirmed each of them. For further confirmation, we posted them to corresponding application's bug-tracking system, and every category has issues that have already been confirmed by application developers. 53 API misuses have been confirmed, and 29 already fixed in their code repositories based on our bug reports. None of our reports has been denied.

Only 3 out of these 428 API misuses coincide with the 64 performance problems listed in Table 2.6 and fixed in Section 4.4. This is because most of these 428 cases do not reside in the 40 problematic actions that we have identified as top issues in our profiling. However, they do cause unnecessary performance loss, which could be severe under workloads that differ from those used in our profiling.

In sum, the above results confirm our previously identified issues, and furthermore indicate that simple API misuses are pervasive across even the latest versions of these ORM applications. Yet, there are many other types of API misuse problems discussed in Section 2.4.1 that cannot be detected simply through regular expression matching and will require future research to tackle.

2.7 Discussion

In this section, we summarize the lessons learned and highlight the new research opportunities that are opened up by our study.

Improving ORM APIs Our study shows that many misused APIs have confusing names, as listed in Table 2.7, but are translated to different queries and have very different performance. Renaming some of these APIs could help alleviate the problem. Adding new APIs can also help developers write well-performing code without hurting code readability. For example, if Rails provides native API support for taking union of two queries' results like Django [16] does, there will be fewer cases of inefficient computation, such as those discussed in Section 2.4.1. As another example, better rendering API supports could help eliminate inefficient partial render problem discussed in Section 2.4.1. To our best knowledge, no ORM framework provides this type of rendering support.

Support for design and development of ORM applications Developers need help to better understand the performance of their code, especially the parts that involve ORM APIs. They should focus on not only loops but ORM library calls (e.g., joins) in performance estimation, since these calls often execute database queries and can be expensive in terms of performance. Building static analysis tools that can estimate performance and scalability of ORM code snippets will alleviate some of the API misuses. More importantly, this can help developers design better application functionality and interfaces, as discussed in Section 2.4.3.

Developers will also benefit from tools that can aid in database design, such as suggesting fields to make persistent, as discussed in Section 2.4.2. While prior work focuses on index design [4], little has been done on aiding developers to determine which fields to make persistent. As the ORM application already contains information on how each object field is computed and used, this provides a great opportunity for program analysis to further help in both aspects.

Compiler and runtime optimizations While some performance issues are related to developers' design decisions, we believe that others can be detected and fixed automatically. Previous work has already tackled some of the issues such as pushing computation down to database through query synthesis [58], query batching [57, 83], and avoiding unnecessary data retrieval [56]. There are still many automatic optimization opportunities that remain unstudied. This ranges from checking for API misuses, as we discussed in Section 2.6, to more sophisticated database-aware optimization techniques to remove unnecessary computation (Section 2.4.1) and inefficient queries (Section 2.4.1).

Besides static compiler optimizations, runtime optimizations or trace-based optimization for ORM frameworks are further possibilities for future research, such as automatic pagination for applications that render many records, runtime decisions to move computation between the server and the DBMS, runtime decisions to switch between lazy and eager loading, and runtime decisions about whether to remove certain expensive functionalities as discussed in Section 2.4.3. Automated tracing and trace-analysis tools can help model workloads and workload changes, which can then be used to adapt database and application designs automatically. Such tools will need to understand the ORM framework and the interaction among the client, server, and DBMS.

Generalizing to other ORM frameworks Our findings and lessons apply to other ORM frameworks as well. The database design (Section 2.4.2) and application design trade-offs (Section 2.4.3) naturally apply across ORMs. Most of the API use problems (Section 2.4.1), like unnecessary computation (UC), data accessing (ID, UD), and rendering (IR), are not limited to specific APIs and hence are general. While the API misuses listed in Table 2.7 may appear to be Rails specific, there are similar misuses in applications built upon Django ORM [16] as well: `exists()` is more efficient than `count>0` (①); `filter().get()` is faster than `filter().first` (②); `clear _ordering(True)` is like `except(:order)` (③); `all.update` can batch updates (④); `len()` is faster than `count()` with loaded arrays (⑤); `only()` is like `pluck()`(⑥); `aggregate (Sum)` is like `sum` in Rails (⑦); `union` allows two query results to be unioned in database (⑧); `get_or_create` is like `find_or_create_by` in Rails (⑨). We sampled 15 issue reports each from top 3 popular Django applications on GitHub. As shown below, these 45 performance issues fall into the same 8 anti-patterns our 140 Rails issue reports fall into:

	IC	UC	ID	UD	MF	MI	DT	FT
Redash [37]	2	3	6	0	0	0	2	2
Zulip [48]	2	5	2	1	0	2	1	2
Django-CMS [17]	0	9	3	0	1	0	1	1

CHAPTER 3
POWERSTATION: DETECTING AND FIXING PERFORMANCE INEFFICIENCIES

3.1 Overview

3.2 Performance Anti-Patterns

PowerStation currently tackles six performance anti-patterns. While these patterns have been observed in previous work [95, 99, 54] from real-world Rails applications, they have not been systematically detected and fixed before—three anti-patterns (RD, CS, and IA below) were automatically detected in three different frameworks; and we are unaware of prior work that performs automatic patching.

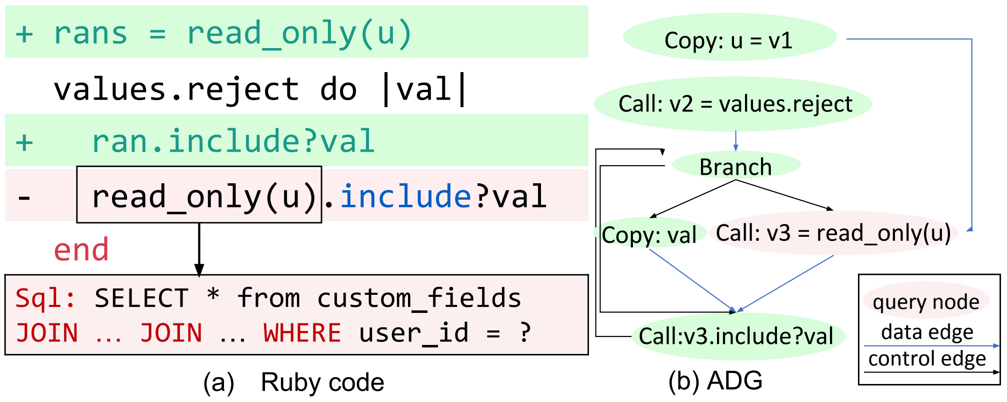
Loop invariant queries (LI) [99]. A query is repeatedly issued in every iteration of a loop to load the *same* database contents. In the real-world example shown in Figure 3.1a, hoisting the query out of the loop can speed up the application by more than 10× [39].

Dead store queries (DS) [99]. SQL queries are repeatedly issued to assign the same memory object with different database contents, without any use of the memory object in between, making all but the last query unnecessary.

Unused data-retrieval queries (RD) [54, 99]. Data is retrieved from the database but never used in the program, making the corresponding data transfer and query execution unnecessary.

Common sub-expression queries (CS) [95]. Queries with common sub-expressions are issued, causing unnecessary re-computation.

API misuses (IA) [99]. Different ORM APIs retrieve the same contents from the database, but they differ drastically in terms of performance. For example, the two Rails code snippets in Figure 3.2 both check if a `user` owns any blog posts. However, they use



```
+ rans = read_only(u)
  values.reject do |val|
+   ran.include?val
-   read_only(u).include?val
  end
Sql: SELECT * from custom_fields
JOIN … JOIN … WHERE user_id = ?
```

Figure 3.1: Loop invariant query from Redmine (the code checks which `val` in `values` list belongs to user `u`'s read-only fields)

```
Ruby: if user.blogs.count > 0
Sql:  select count(*) from blogs where user_id = ?
Ruby: if user.blogs.exist?
Sql:  select 1 from blogs where user_id = ? limit 1
```

Figure 3.2: API misuse from Onebody (the upper code is less efficient than the lower code)

different APIs, `count` versus `exist`, that are translated to different SQL queries by Rails: `select count` vs. `select limit 1`. The former query scans all records in the `blogs` table with specified `user_id`, counts the number of records, and checks (in the Ruby application) if the count is greater than 0. The latter query returns immediately when it finds one record with the specific `user_id`, which can reduce query execution time by 1.7× comparing to the former.

Inefficient data rendering (IR) [99]. While rendering a list of objects, helper functions are often used to render a partial view for one object at a time, with much redundant computation repeated for every object. For example, the HTML in Figure 3.4b is generated line by line by repeated invocations of `link_to` with much redundancy across lines. Such inefficiency is particularly severe when there are many objects to render. Consequently, it could become a scalability bottleneck when the objects need to be first retrieved from database.

We find these six anti-patterns to be prevalent even in well-developed applications as developers are often unaware of what database queries are issued due to the ORM abstraction. Such queries also cannot be optimized by traditional Ruby compilers as they treat ORM APIs as black boxes (nor database engines as they can only observe the queries issued by the application). We next explain how PowerStation can detect such patterns.

3.3 PowerStation's static analysis

PowerStation's static analysis contains two components. The first takes in Rails source code and generates a database-aware program dependency graph for every action,[1] which we refer to as the action dependency graph (ADG). The second component takes in the ADG, identifies performance anti-patterns, and synthesizes fixes. We anticipate extending PowerStation to tackle other ORM-related performance issues in the future.

3.3.1 Database-aware Static Analysis Framework

PowerStation's static analysis framework goes through the following steps to generate ADG from Ruby on Rails source code.

Pre-processing. PowerStation performs inter-procedural analysis on the source code by statically inlining all function calls. It also inlines callbacks such as `ActiveRecord` validations invoked by this action. Since Ruby is dynamically typed, PowerStation performs type inference [66] to statically determine variable types.

Program dependency graph (PDG) generation. PowerStation generates a PDG for every controller action, which is the entry function that eventually produces a webpage. It uses JRuby to parse the pre-processed source code, and then builds the PDG from JRuby's intermediate representation (IR) as this IR nicely captures high-level Ruby semantics via instructions and operands. As illustrated in Figure 3.1b, every node n in the PDG represents

1. An action is a member method of a Ruby controller class. When a web application receives a request, a corresponding action will execute to respond to the request.

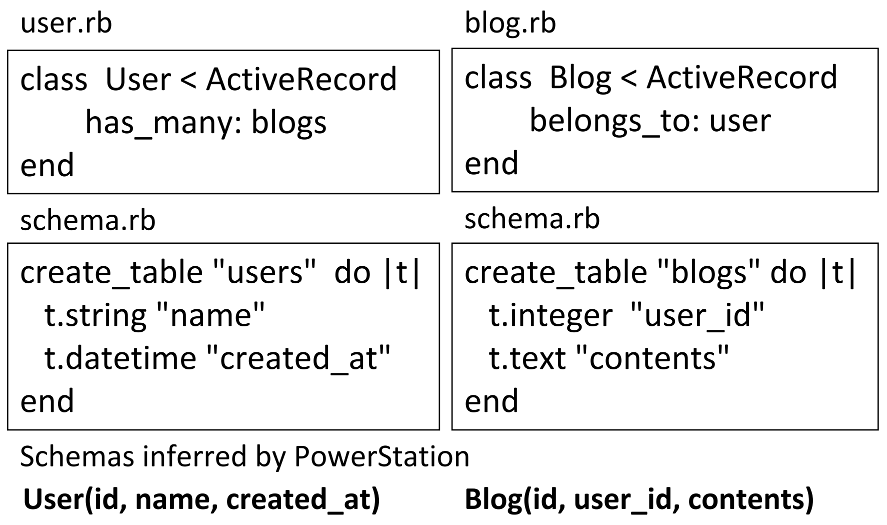
a statement in the JRuby IR. Every edge e in PDG represents either control dependency or data dependency. A data-dependency edge $n_1 \rightarrow n_2$ indicates that the output object o of n_1 is used by n_2 without other statements overwriting o in between.

Database-aware ADG Generation. PowerStation then enhances the PDG generated above in three ways to create the ADG: (1) changing and splitting some nodes to become Query nodes; (2) annotating every Query node with the database table and fields that are read or written; (3) annotating every outgoing data-dependency edge of a Query node with the exact field(s) that are used.

To accomplish this, PowerStation first analyzes every model class that extends the Rails `ActiveRecord` interface to determine all the database tables in the application and the association relationship among them. For example, analyzing the model classes illustrated in Figure 3.3, PowerStation identifies the `users` table corresponding to the `User` class and similarly for the `Blog` class, and that these two models have a one-to-many relationship, i.e., each instance of `User` may own multiple instances of `Blog`. Second, PowerStation analyzes the `schema.rb` file to determine how many fields each table contains. For example, parsing the `schema.rb` snippet in Figure 3.3, PowerStation infers the schemas of the `users` and `blogs` tables as shown in the bottom of the figure.

Third, PowerStation identifies queries from three sources: (1) explicit invocations of Rails `ActiveRecord` Query APIs, such as `exist?`, `reload`, `update`, `destroy`, etc; (2) implicit queries generated by Rails to access object fields, e.g., o_1`.`o_2, where the class of o_1 and the class of o_2 are associated model classes (e.g., `user.blogs` would incur a query to retrieve records in `blogs` table that are associated with the specific `user` record in `users` table); (3) embedded SQL queries through `Base.connection.execute`. Any query identified above is represented as a Query node in the ADG.[2]

2. At run time, multiple such queries could be composed by ORM into one SQL query. Such query chaining does not affect PowerStation's analysis.


user.rb

```
class  User < ActiveRecord
      has_many: blogs
end
```

blog.rb

```
class  Blog < ActiveRecord
      belongs_to: user
end
```

schema.rb

```
create_table "users"  do |t|
  t.string "name"
  t.datetime "created_at"
end
```

schema.rb

```
create_table "blogs" do |t|
  t.integer  "user_id"
  t.text "contents"
end
```

Schemas inferred by PowerStation

User(id, name, created_at) **Blog(id, user_id, contents)**

Figure 3.3: Analyzing table schemas

3.3.2 Detecting and Fixing Anti-patterns

Loop invariant queries. PowerStation first identifies all query nodes inside loop bodies in ADG. For each such node n, it checks the incoming data-dependency edges of n. If all of these edges start from outside the loop L where n is located, then n is identified as a loop-invariant query, such as "Call: v3..." in Figure 3.1b. To fix this, PowerStation inserts a new Assign statement before the start of the loop, where a newly created variable v gets the return value of the loop-invariant query, and replaces every invocation of the loop-invariant query inside loop L with v.

Dead store queries. PowerStation checks every ADG node that issues a reload query, i.e., `o.reload`, and checks its out-going data-dependency edge. If there is no such edge, i.e., the reloaded content is not used, then the query is marked as a dead-store query that is deleted by PowerStation.

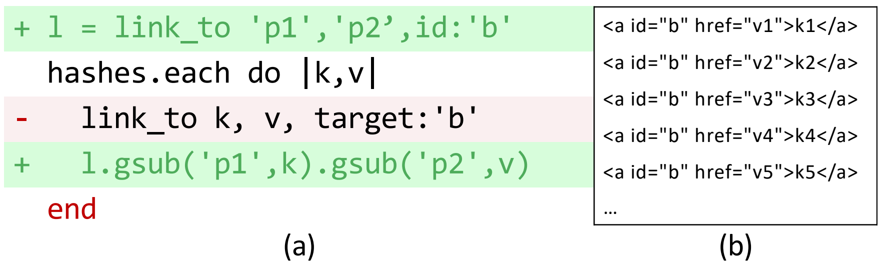
Unused data retrieval queries. For every Read query node n in the ADG, PowerStation first computes the database fields loaded by n that are used subsequently. This is the union of the *used fields* associated with every out-going data-dependency edge of n. PowerStation then checks if every loaded field is used. For every unused field, PowerStation either deletes n, if none of the fields retrieved by n are used, or adds field selection `.select(:f1, :f2, ...)` to the original query in n so that only used fields `f1`, `f2` are loaded.

Common sub-expression queries. PowerStation checks every query node q_0 in ADG to see if q_0 has out-going data-dependency edges to at least two query nodes q_1 and q_2 in the same control path. If that is the case, then by default, Rails would issue at least two SQL queries that share common sub-expression q_0 at run time, one composing q_0 and q_1 and one composing q_0 and q_2, with the latter unnecessarily evaluates q_0 again. This can be optimized by changing the query plan and caching the common intermediate result for reuse [95]. Doing so requires issuing raw SQL commands that are currently not supported by Rails `ActiveRecord` APIs.

API Misuses. PowerStation uses regular expression matching to find inefficient API misuses as in previous work [99]. Since these API mis-use patterns are simple, PowerStation also synthesizes patches for each API mis-use pattern through regular expressions.

Inefficient rendering. PowerStation checks every loop in the ADG to see if it iterates through objects returned by queries and contains a Rails view helper function such as `link_to` in every loop iteration. If so, PowerStation identifies the code as having the inefficient rendering problem. To fix this, PowerStation hoists the helper function out of the loop, assigns its result to a newly created variable, and replaces the original helper function in the loop with `gsub` (a string substitution function) on the newly created variable, as shown in Figure 3.4. Doing so removes the redundant rendering that is performed on every loop iteration in the original code.

Discussion. Like other code refactoring tools, PowerStation currently suggests fixes to the user rather than deploying them automatically. This is important for Dead Store and Unused Data cases, since the PowerStation-suggested fixes would change application semantics if the retrieved data is used in multiple actions. For the other cases, however, PowerStation's suggested fixes do preserve program semantics.


```
+ l = link_to 'p1','p2',id:'b'
  hashes.each do |k,v|
-   link_to k, v, target:'b'
+   l.gsub('p1',k).gsub('p2',v)
  end
```

```
<a id="b" href="v1">k1</a>
<a id="b" href="v2">k2</a>
<a id="b" href="v3">k3</a>
<a id="b" href="v4">k4</a>
<a id="b" href="v5">k5</a>
…
```

Figure 3.4: Fix for inefficient rendering (`gsub` is a string substitution API, replacing its first parameter with the second)

3.4 PowerStation IDE integration

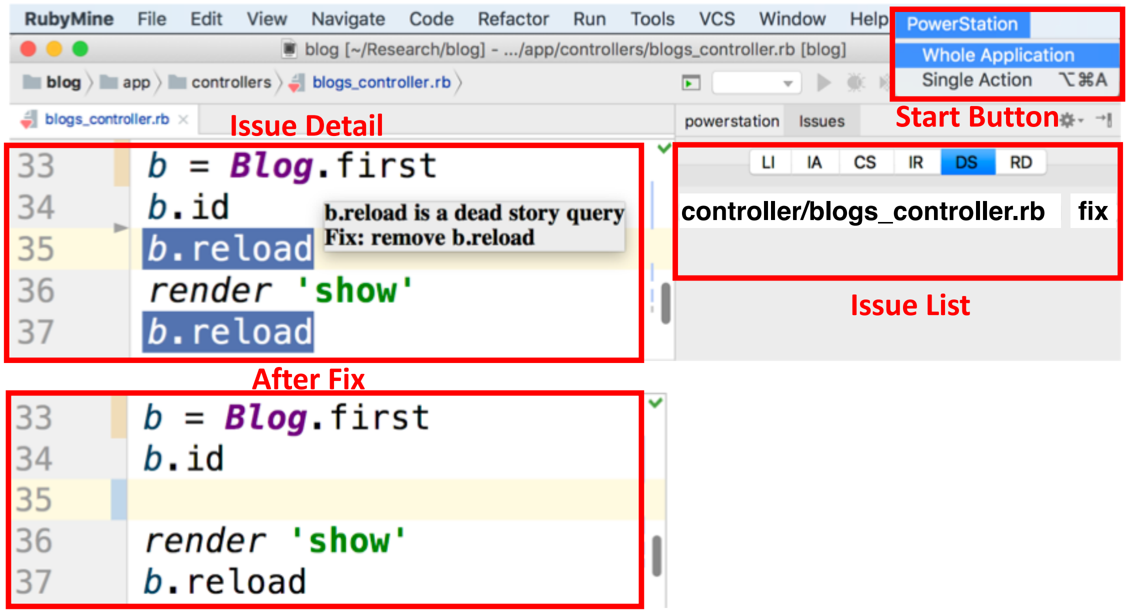
3.4.1 PowerStation IDE Plugin Features

We have implemented PowerStation as an IDE plugin for RubyMine [43], a popular IDE for Ruby on Rails. A screenshot of PowerStation is shown in Figure 3.5. By pressing the "PowerStation" button at the top of RubyMine, users can choose an analysis scope, "Whole Application" or "Single Action," and launch PowerStation analysis accordingly. Our website includes a tutorial [34].

Issues list. The right panel, as highlighted in Figure 3.5, lists all the inefficiencies detected by PowerStation, each represented by a button displaying the file where the inefficiency is located. By default, all the inefficiencies found in the project are listed. Users can also choose to display inefficiencies of a particularly type as shown in Figure 3.5—loop invariant queries (LI), dead store queries (DS), unused data retrieval queries (RD), common sub-expression queries (CS), API misuses (IA), and inefficient rendering (IR).

Issues highlight. Clicking the file button in the issue list will navigate users to the corresponding file in the editor, with the inefficient code highlighted. Hovering the cursor over the highlighted code will display the reason for highlighting, as shown in Figure 3.5.

Issue fix. Clicking the "fix" button next to each issue in the issue list will pop up window asking the user whether she wants PowerStation to fix the issue. If so, PowerStation will synthesize a fix as discussed in Section 3.3, and display the fixed code in the editor panel.


RubyMine File Edit View Navigate Code Refactor Run Tools VCS Window Help
PowerStation
Whole Application
Single Action ⌥⌘A
Start Button
blog [~/Research/blog] - .../app/controllers/blogs_controller.rb [blog]
blog app controllers blogs_controller.rb
blogs_controller.rb
Issue Detail
powerstation Issues
LI IA CS IR DS RD
Issue List
After Fix

Issue Detail:

```
b = Blog.first
b.id
b.reload
render 'show'
b.reload
```

b.reload is a dead story query
Fix: remove b.reload

Issue List:

controller/blogs_controller.rb fix

After Fix:

```
b = Blog.first
b.id

render 'show'
b.reload
```

Figure 3.5: Screenshots of PowerStation IDE Plugin

At that point, the original “fix” button becomes an “undo” button, allowing users to revert the fix if needed.

3.4.2 Implementation

We used the APIs provided by the IntelliJ Platform like `ToolWindow` and `JBTabbedPane` to create the PowerStation issues list.

Highlighting the selected inefficiency is straight-forward using the IntelliJ API `HighlighterLayer`, given file name and line number provided by PowerStation static analysis.

For every anti-pattern, PowerStation prepares a string template that explains the inefficiency and the fix strategy, such as “* is a dead store query. Fix: delete *.” for a dead-store query (Figure 3.5). This string is instantiated with program variables and expressions output from PowerStation static analysis, and displayed using the IntelliJ API `FileDocumentManager`.

Finally, IntelliJ API `FileEditorManager`, `TextRange`, and `Document` are used to insert, replace, and delete source code in the editor panel.

3.5 Evaluation

PowerStation can be downloaded from IntelliJ plugin repository [18] and easily installed in RubyMine.

We evaluated PowerStation using the latest versions of 12 open-source Rails applications, including the top two popular applications on Github from 6 categories: Forum, Collaboration, E-commerce, Task-management, Social Network, and Map. As shown in Table 3.1, PowerStation can automatically identify 1221 inefficiency issues and generate patches for 730 of them (i.e., all but the common sub-expression pattern). We randomly sampled and examined half of the reported issues and the suggested fixes, and found no false positives. Due to the limited resource and time, we reported 433 issues with 57 of them already confirmed by developers (none has been denied). PowerStation static analysis is fast, taking only 12–625 seconds to analyze the entire application that ranges from 4k to 145k lines of code in our experiments on a Chameleon instance with 128GB RAM and 2 CPUs. Developers can also choose to analyze one action at a time, which usually takes less than 10 seconds in our experiments.

3.6 Conclusion and Future Work

PowerStation is a new tool that automatically detects and fixes a large set of ORM-related performance issues that are both common and severe in database-backed web applications. Its integration with RubyMine provides an easy way for Rails developers to avoid making performance-degrading mistakes in their programs. We have used PowerStation to identify and fix many performance-related issues in real-world applications, and will extend PowerStation tackle further performance anti-patterns as future work.

Table 3.1: Inefficiencies detected by PowerStation in 12 apps

App.	Loop Invariants	Unuse Data	Common Sub-expr	API Misuses	Inefficient Render	SUM
Ds	0	16	106	85	0	207
Lo	0	2	0	45	5	52
Gi	0	14	92	23	1	130
Re	0	11	101	59	0	171
Sp	0	22	0	20	0	42
Ro	0	3	0	11	0	14
Fu	0	12	15	2	1	30
Tr	0	23	30	30	1	84
Da	1	55	36	57	0	149
On	0	17	39	76	0	132
FF	0	24	12	4	5	45
OS	0	89	60	16	0	165
SUM	1	288	491	428	13	1221

CHAPTER 4

PANORAMA: VIEW-CENTRIC PERFORMANCE OPTIMIZATION

4.1 Overview

In this chapter, we present a framework called Panorama that provides a view-centric and database-aware analysis for web developers to understand and optimize their database-backed web applications. Panorama currently targets applications written using the Ruby on Rails framework, and makes three major contributions as illustrated in Figure 4.1.

Panorama provides a view-centric estimator that helps developers understand the data-processing cost behind every HTML tag. Panorama both dynamically monitors database query performance using the test workload, statically estimates data processing complexity independent of any specific workload, and carefully attributes the cost to every HTML tag through its cross-stack dependency analysis. The details will be presented in Section 4.3.

Panorama provides a view-aware performance optimizer that helps developers carry out view-changing code refactoring to improve performance. Panorama suggests a variety of refactorings that (1) change the manner of content rendering (i.e., pagination or asynchronous loading); or (2) change the accuracy of the rendered contents (i.e., approximation); or (3) remove certain web-page contents from rendered contents. Through static program analysis, Panorama not only identifies opportunities for applying such refactoring, but also automatically suggests patches that complete such refactoring, often involving modifications to multiple files in model, view, and controller components. We present the details in Section 4.4.

Panorama provides a unique interface for developers to effectively exploring different web-page designs with different performance-functionality trade-offs. Instead of separately presenting profiling information and refactoring suggestions, Panorama integrates them in

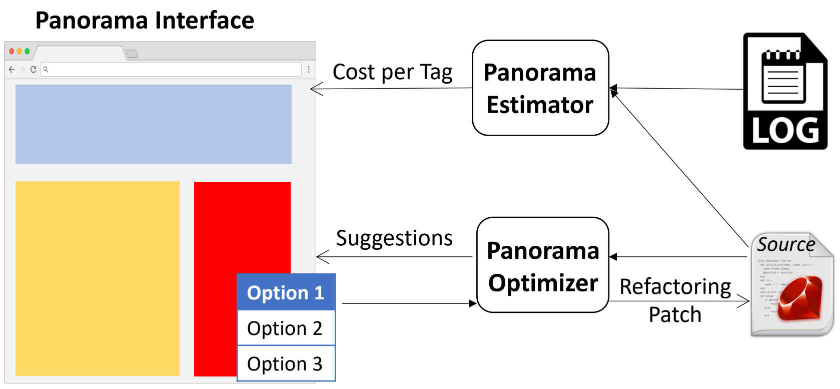


Figure 4.1: Panorama overview

the web browser—while testing a page of their web applications, the data processing cost for each HTML tag is presented as a heat map in the browser. Developers can right click on each HTML tag to see the different view-changing options for performance enhancement; they can choose any option and immediately see an updated web page with an updated heat map in the browser, with all code refactoring automatically done by Panorama in an accompanying Ruby editor.

We evaluated Panorama on 12 popular open-source Ruby on Rails applications. Panorama statically identifies 140 performance-enhancing opportunities through view changes. We randomly sampled 15 view changes suggested by Panorama and found that by applying the patches automatically generated by PowerStation, these 15 view changes speed up end-to-end page load time by 4.5× on average (39 maximum), using database workloads that are similar to those used in real-world deployments. We believe the benefits will increase with even larger workloads. Furthermore, we conducted a thorough user study with 100 participants from Amazon Mechanical Turk. The study shows that web pages with these view changes are considered as similar or better than the original web pages in most cases, with more users preferring the design suggested by Panorama than the original ones. This user study result, as well as the fact that these optimizations save computation resources on web servers and database servers, justify the need for developers to explore the performance-functionality trade-off space in web application design, with Panorama being a first step towards that goal.

4.2 Static analysis framework

Panorama leverages a database-aware static analysis framework for Rails applications that we briefly describe below.

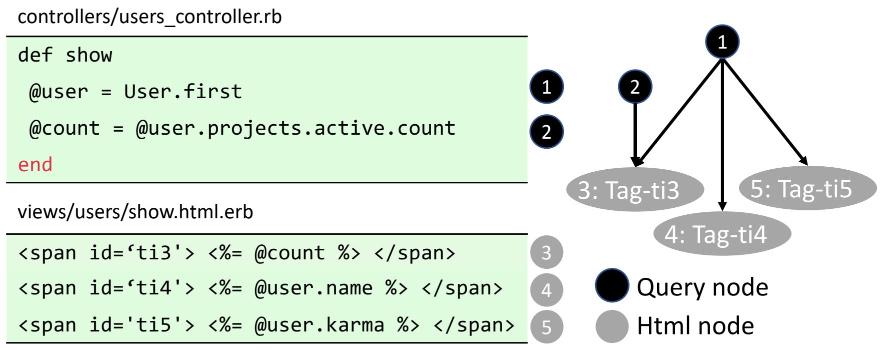
4.2.1 Action dependency graph

Panorama's static analysis centers around the action-dependency graph (ADG) that is constructed for each controller action. Figure 4.2 shows an example of ADG.

An ADG is a database-aware extension of the traditional program-dependence graph (PDG) [65]. Every node n in the ADG represents an intermediate representation (IR) statement in the corresponding action's JRuby [28] IR. Every edge e represents either control dependency or data dependency. Edges shown in Figure 4.2 all represent data dependencies.

In contrast to the PDG, every node in the ADG that issues a SQL query is associated with a query tag in ADG, such as node ① and node ② in Figure 4.2. Information about SQL queries that are issued and the tables they referenced are determined by analyzing `ActiveRecord` function calls and recorded in the ADG.

Since view files may also contain Ruby code to process or render data, they are also analyzed during the ADG construction. Specifically, for every action, like `user/show` in Figure 4.2, its corresponding view file is identified based on an explicit `render` statement or implicit file-name matching (as in Figure 4.2). The corresponding view file is then parsed, with all Ruby code embedded inside `<% ...%>` extracted and inlined as part of the ADG, like the three statements inside `show.html.erb` and the ADG shown in Figure 4.2.


controllers/users_controller.rb

```
def show
 @user = User.first                        1
 @count = @user.projects.active.count      2
end
```

views/users/show.html.erb

```
<span id='ti3'> <%= @count %> </span>        3
<span id='ti4'> <%= @user.name %> </span>    4
<span id='ti5'> <%= @user.karma %> </span>   5
```

Figure 4.2: Excerpt of an Action Dependency Graph

4.2.2 Annotating the view component

In order for Panorama to attribute performance data correctly to each HTML tag, Panorama pre-processes every view file in the input web application to assign every HTML tag a unique ID. That is, for every tag `<tag>` that does not already have an ID, Panorama turns it into `<tag id = ti>`, where `ti` is a unique ID, as shown in the `<span>` tags of Figure 4.2. The current prototype of Panorama does not handle HTML tags that are programmatically generated by JavaScript code.

As mentioned, the view file has Ruby code embedded within it. For every node in ADG whose source code is in a view file, Panorama identifies its inner-most surrounding HTML tag and associates it with the corresponding tag ID. Panorama also checks whether its corresponding content is rendered or not by analyzing the HTML, and assigns an `is_rendered` property accordingly. This information will help Panorama attribute data processing cost to each HTML element and identify alternative view designs, as we will explain in later sections.

4.3 PowerStation view-centric cost estimator

There are two key tasks in PowerStation's cost estimation. First, given an HTML tag, PowerStation determines which database queries are executed to generate the data that is rendered through that tag (we refer to them as contributing queries). Second, for each

HTML tag, PowerStation measures the data processing cost needed to render it.

While a web page's load time consists of client side rendering time, network communication between client and server, computation time on the server, and database query cost, PowerStation's estimator currently focuses on database query cost, as query time often contributes a significant portion of the page load time. This is particularly true as the data size increases, and large query results lead to even more computation and rendering time. Query time/complexity is also difficult for developers to estimate, particularly given the ORM abstraction. As future work, we will incorporate other profiling tools to measure the performance of the client code [9], network, and server computation as part of PowerStation.

4.3.1 Identifying contributing queries

PowerStation identifies contributing queries for an HTML tag by statically analyzing control and data dependencies in the ADG. Given an HTML tag in a view file, PowerStation first identifies all ADG nodes N that contain the tag's unique ID — these nodes contain the Ruby code embedded in the HTML tag. Then, PowerStation traces backward along the ADG edges to identify all query nodes that any node in N has control or data dependence upon. All these queries, each identified by its ADG node ID and Ruby source code location, are considered as contributing queries of this HTML tag. For example, in Figure 4.2, tracing dependency edges backward from node ③, which corresponds to HTML tag with id `ti3`, will identify two contributing queries, nodes ① and ②.

PowerStation further conducts forward dependency checking in the ADG to see how many other HTML tags each query node contributes to. This number can be used as a weight in computing the data processing cost of an HTML tag — if a query result is used to generate k HTML tags (e.g., the query node ① contributes to three HTML tags in Figure 4.2), we could attribute $1/k$ of the query cost to each HTML tag if the web developers choose so (while by default PowerStation attributes the complete query cost to each tag).

4.3.2 Cost analysis

PowerStation offers two modes of cost estimation with or without relying on testing workload.

Dynamic profiling

If a testing workload is available, PowerStation will measure the cost of each contributing query during a testing run. However, using the query execution log from the backend database engine as the testing workload does not work — the database engine has no knowledge about frontend Ruby and HTML code, and hence does not allow PowerStation to connect the statement in the web application that issues the query and the HTML tag uses the query result.

PowerStation instead conducts its profiling through a hook API provided by Rails infrastructure, `ActiveRecordQueryTrace`. This API allows its hooked code to be called before and after issuing each SQL query. Using this mechanism, PowerStation logs the amount of time of each query and the line of source code that issues this query during the testing run, and attributes the time to the corresponding HTML tags using contributing query analysis as discussed above.

Static estimation

Since a bottleneck-exposing workload may not be available during in-house testing, PowerStation also uses static analysis to estimate the potential data-processing cost (in terms of its data complexity) to render each HTML tag. For ease of estimation, PowerStation assumes that all tables in the database have the same size D. Then, for each contributing query, PowerStation estimates its complexity (i.e., how its execution time might increase with D) by considering: (1) the number of times this query might be issued, and (2) time taken to execute one query instance.

To estimate the first factor, PowerStation analyzes loops. If the query Q is not contained in any loop or is only contained by a loop whose iteration number does not increase with D, which we refer to as a *bounded loop*, PowerStation then considers Q to be executed for a constant number of times. Otherwise, PowerStation considers Q to be executed for D^k times, with k being the number of *unbounded* loops containing Q. To identify the *unbounded* loops, PowerStation analyzes the bound variable of all loops that contain Q. If the loop iterates through a set of records returned by an *unbounded* database query, PowerStation considers the loop to be unbounded. Specifically, in Rails, a query is *unbounded* in all but the following three cases: (1) it always returns a single value, like a `SUM` query; (2) it always returns a single record by selecting on primary key; (3) it always returns a bounded number of records using the `LIMIT` keyword.

To estimate the second factor, PowerStation first identifies all the query operators inside the query Q. For example, for the query node ② in Figure 4.2, PowerStation would identify three query operators from the `@user.issues.active.count` statement: a `SELECT` to get `issues`, another `SELECT` to get `active`, and finally a `count` operator. For most operators, we estimate its execution complexity to be $O(D)$. There are a few exceptions: we consider the complexity of a `JOIN` operator to be $O(D^2)$, and the complexity of an operator that explicitly uses index, such as `find` and `find_by_id`, to be constant.

Putting these two factors together gives PowerStation the complexity estimation for one contributing query Q. For example, the estimated complexity of the query node ② in Figure 4.2 is $O(D^3)$. If it is enclosed in an unbounded loop, its complexity would increase to $O(D^4)$.

PowerStation could choose to deliver the above performance information using either a detailed text description or a numeric score. The current prototype uses the latter: it uses the highest complexity among all contributing queries as the complexity score of an HTML tag. For example, in Figure 4.2, the complexity score of HTML node ③ is 3, based on the $O(D^3)$ complexity estimated for query node ②, and the complexity scores of HTML node ④ and ⑤ are both 1, based on the $O(D)$ complexity estimated for query node ①.

Of course, this is just a best-effort estimation from PowerStation. There are several potential sources of inaccuracy that can be improved by future work. For example, some database tables may be much larger than others, which we do not consider; the database can also lower the query complexity than our estimation due to query optimization.

4.4 Panorama view-aware optimization

PowerStation suggests three categories of view-changing code refactoring to improve page-load time:

1. Display the same contents in a different style, such as pagination and asynchronous loading.
2. Display the same contents but with a different accuracy.
3. Remove a subset of contents from display.

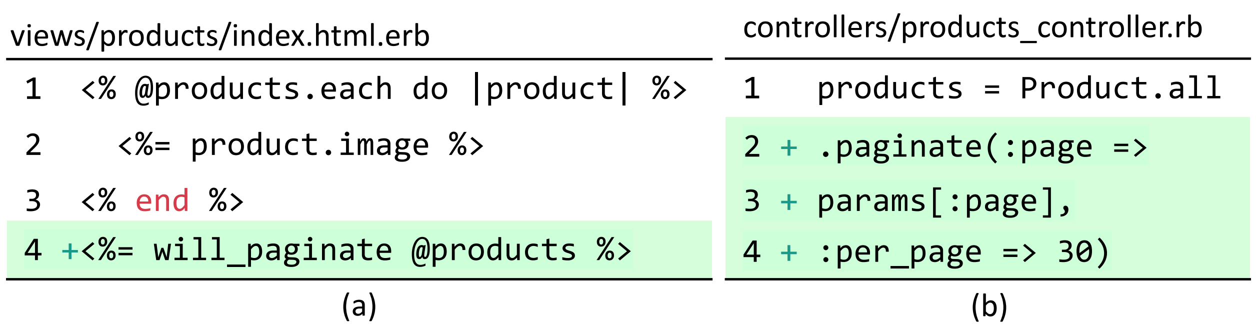
These code refactorings can be applied for different types of HTML tags, and complement each other.

Panorama code refactoring works independently from Panorama cost estimator. As we will see in Section 4.5, Panorama interface will contain both features and help developers make informed refactoring decisions.

4.4.1 Display-style change: pagination

Many web pages are designed to display *all* database records satisfying certain conditions. When the database size grows, such pages will take an increasingly more time to load, and eventually become unresponsive.

A widely used solution to this problem, called *pagination*, is to display only a fixed number of records in a page and allow users to navigate to other pages for more records.


views/products/index.html.erb

```
1  <% @products.each do |product| %>
2    <%= product.image %>
3  <% end %>
4 +<%= will_paginate @products %>
```

(a)

controllers/products_controller.rb

```
1   products = Product.all
2 + .paginate(:page =>
3 + params[:page],
4 + :per_page => 30)
```

(b)

Figure 4.3: Refactoring so that the paginated page displays 30 items at a time rather than the full list

Although pagination is widely used in practice, there are still many cases where it is not used — 14 out of 140 real-world performance issues sampled by a previous study [100] are due to lack of pagination — either because developers are unaware of pagination, or because they did not anticipate the data size will become a performance problem. Therefore, we design Panorama to automatically identify pagination opportunities and conduct corresponding refactoring for developers.

Identifying opportunities

To identify these opportunities, PowerStation checks each loop in the program for: (1) whether the loop iterates through the result of an unbounded query; and (2) whether each loop iteration leads to some content being rendered in an HTML tag. If a loop passes both checks, the corresponding HTML tag will be reported as a pagination candidate.

For the first check, PowerStation locates the array variable that a loop iterates through, like `@products` in the loop shown in the left column of Figure 4.3, and then checks the data-flow edges in ADG to determine whether this variable is produced by an unbounded database query, as defined in Section 4.3.2. For example, the ADG would show that `@products` is the result of `Product.all` on Line 1 of in Figure 4.3b, and `Product.all` will be translated to an unbounded database query at run time.

For the second check, PowerStation searches for an ADG node n_v that is associated with an HTML tag and an `is_rendered` property inside the loop body (how to compute `is_rendered` is introduced in Sec. 4.2.2). If n_v is found, like Line 2 in Figure 4.3a, the HTML tag associated with n_v is identified as a pagination candidate.

Generating patches

To carry out the refactoring, PowerStation performs two changes to the source code using the `will_paginate` library [47]. First, in the controller, PowerStation adds a `.paginate` call right after the code statement where the to-be-rendered database records are retrieved, like Line 2, 3 and 4 in Figure 4.3b. The constant there, which is configurable and 30 by default, determines how many records will be shown on every page. Second, in view, PowerStation adds a `<=% will_paginate @products %>` statement right after the loop that renders these database records, as illustrated in Line 5 in Figure 4.3a. The `will_paginate` call inserts a page navigation bar into the web page, allowing users to navigate to remaining records after seeing the records displayed on the current page.

4.4.2 Display-style change: asynch-loading

Asynchronous programming is widely used to support low-latency interactive software [80, 79, 72]. For web applications, when there is an HTML tag that takes much longer time to render than other tags on the same page, we can instead compute and render the slow tag asynchronously, allowing users to see other parts of the web page more quickly.

For example, Discourse is a forum application. Its `topics/show` page mainly lists all the `posts` that belong to a topic. At the bottom of that page after the listing of all the `posts`, a list of `suggested topics` that are related to this topic are displayed. In an issue [14], users complained that this page is slow to load no matter a topic contains many or few posts. Developers then realized that the query to retrieve suggested topics is hurting the page-load time. Making things worse, these suggested topics are not the main interests of this page and often are not seen by users, as they are placed below all the posts and require users to scroll down to the bottom of the page to see. Consequently, developers created a patch that defers the display of `suggested topics` until all other content on the page is displayed.

Identifying opportunities

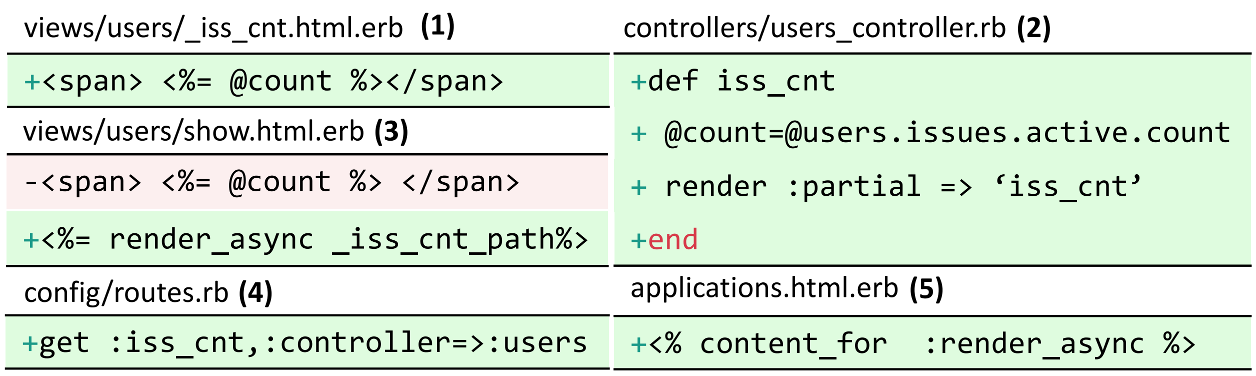
Conceptually, every HTML tag can be computed and rendered asynchronously. We only need to pay attention to two issues.

First, only tags that are among the slowest on a web page are worthwhile for asynchronous loading. Otherwise, loading an originally fast tag asynchronously does not help shorten the page load time, the PowerStation estimator (Section 4.3) already provides information to help developers make this decision, and hence we do not discuss this issue below.

Second, if too many HTML tags on a web page are rendered asynchronously, the user experience could be greatly degraded. Furthermore, if one HTML tag is rendered asynchronously, other HTML tags may better be rendered asynchronously too if they share a common contributing query. For example, in Figure 4.2, once we decide to load HTML tag ③ asynchronously, ④ and ⑤ will be loaded asynchronously too, as they share a common query ②. PowerStation considers this issue in identifying opportunities for asynchronous loading, and we will describe how this is handled below.

Generating patches

Given an HTML tag e, making its content computed and rendered asynchronously requires multiple changes to the controller and view components of a web application, as illustrated in Figure 4.4: (1) creating a new view file that renders e only, separating e from other tags on the same web page that will still be synchronously loaded; (2) adding a new controller action to compute *all and only* the content needed by e and render the new view file created above, separated from the computation for other tags on the same web page that will still



views/users/_iss_cnt.html.erb **(1)**

```
+<span> <%= @count %></span>
```

views/users/show.html.erb **(3)**

```
-<span> <%= @count %> </span>
+<%= render_async _iss_cnt_path%>
```

config/routes.rb **(4)**

```
+get :iss_cnt,:controller=>:users
```

controllers/users_controller.rb **(2)**

```
+def iss_cnt
+ @count=@users.issues.active.count
+ render :partial => 'iss_cnt'
+end
```

applications.html.erb **(5)**

```
+<% content_for  :render_async %>
```

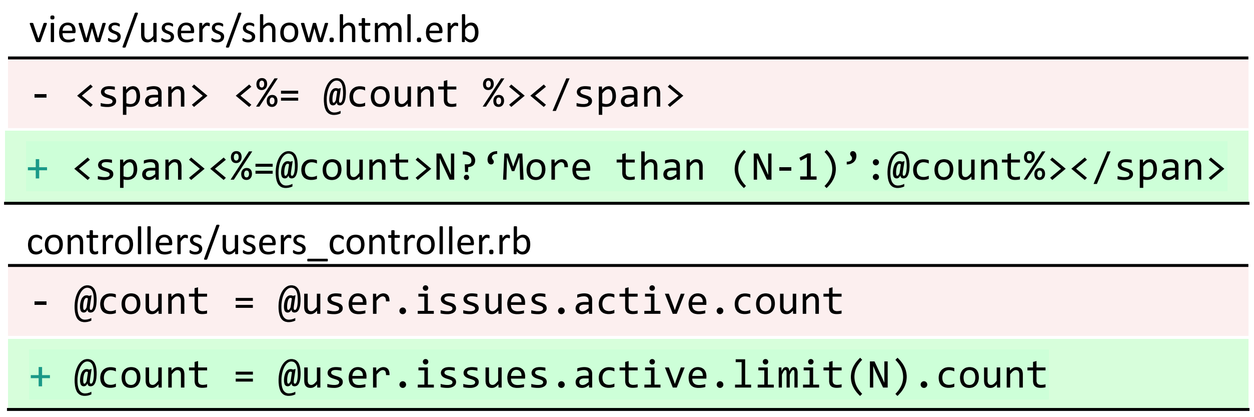
Figure 4.4: Refactoring for asynchronous loading

be carried out synchronously; (3) replacing e in the original view file with an AJAX request and adding a new routing rule so that the AJAX request will invoke the new action in (2) which then renders the view in (1) asynchronously.

The first item is straight-forward to automate. PowerStation simply moves the HTML tag e into a newly created view file, like `_iss_cnt.html.erb` (Figure 4.4(1)), where `iss_cnt` corresponds to the name of the new controller action that PowerStation will generate.

The second item is implemented by PowerStation in three steps. It first identifies all Ruby variables used by e, like `@count` in Figure 4.4(1), and then applies static backward slicing to find all code statements C that are used to compute those variables, like `@count = @user.issues.active.count` in Figure 4.4. Specifically, PowerStation starts from all the ADG nodes associated with the specific HTML-tag ID, and traces backwards in the ADG to identify all nodes inside the corresponding controller that e has control or data dependence on.

PowerStation next applies forward taint analysis to see if any statement $c \in C$ is used to compute any other HTML tag e'. If such an e' is found, there is a dilemma about whether to render e' asynchronously: rendering e' asynchronously could potentially cause many other HTML tags, which share common backward slicing fragments with e', to be rendered asynchronously, and violate the design principle discussed in Section 4.4.2; yet rendering e' synchronously incurs extra overhead as c now needs to be computed twice,



views/users/show.html.erb

```
- <span> <%= @count %></span>
+ <span><%=@count>N?‘More than (N-1)’:@count%></span>
```

controllers/users_controller.rb

```
- @count = @user.issues.active.count
+ @count = @user.issues.active.limit(N).count
```

Figure 4.5: Refactoring for approximation

once for e and once for e'. Hence, PowerStation currently considers e as unsuitable for asynchronous loading if e' exists.

PowerStation finally moves the slice identified earlier to a new controller action, like `iss_cnt` in Figure 4.4(2) (the deletion from the previous controller is not shown for simplicity), and add a rendering statement at the end of the action, like `render :partial => ‘iss_cnt’` in Figure 4.4(2), to render the same content in the same format as the original web application using the newly created view file.

PowerStation conducts the third item leveraging the `render_sync` library [3] to replace the original HTML tag with “`<%= render_async [action] _path %>`” (Figure 4.4(3)), where `render_async` is an API call that issues an AJAX request for the specified `action` using jQuery [27]. PowerStation then adds a new rule into the routing file to connect the AJAX request with the action it just created. As shown in Figure 4.4(4), this new routing rule follows the template “`get :[action], :controller => :[home]`”, where `action` is the name of new action name, and `home` is the controller holding the `action`.

4.4.3 Display-accuracy change: approximation

Approximation is a widely used approach to improving performance and saving server resources [64]. Past database research also proposed approximated queries [75]. However, many techniques require changes to database engines [69] and hence cannot be applied to web application refactoring. PowerStation focuses on approximating aggregation queries whose results are displayed as numeric values on web pages, as such approximation can be simply conducted by refactoring Rails code and easily reasoned about by web viewers.

For example, Redmine [38] is a project collaboration application like GitHub. Its `user/index` page lists all the recent activities of a user, all projects a user is involved in (with pagination), as well as two counts showing how many issues are currently assigned to and have been reported by this user. Although these two numerical counts occupy tiny space on the web page, they can take more time, even more than 1 second, to render than the remaining page, when a user is involved in hundreds of or more issues. One way to keep the page responsive is to set an upper-bound to such a count, like 100, and only shows the count to be "*more than 100*" when it is too big — when a count is too big, users probably does not care about the exact number anyway.

Identifying opportunities

PowerStation iterates through all aggregation, including maximum, minimum, average, and count, queries in the application. For each query, PowerStation checks its corresponding ADG node's out-going data-flow edges to see if the query result is *only* used in HTML-tag rendering. If so, an approximation opportunity is identified for corresponding HTML tag(s). Note that, PowerStation does not suggest approximating an aggregation query if its result affects an HTML tag through control dependency, as that type of approximation may cause program execution to take a different path and hence potentially leads to large deviation from original program behaviors.

Generating patches

An approximation refactoring includes two parts. On the controller side, PowerStation appends a `limit(N)` clause to the end of the aggregation query identified above, with the constant N configured by web developers. On the view side, instead of directly displaying the numeric query result, changes are made depending on the aggregation query type. For a `count` query, PowerStation inserts a conditional statement to check the aggregation result: if the result is smaller than N then the accurate numerical result is displayed, otherwise "more than $N-1$", as shown in Figure 4.5; for an average query, PowerStation adds "`about`" before the numeric query result rendered in the HTML tag; for a maximum or minimum query, PowerStation adds "`at least`" or "`at most`" before the numeric query result.

4.4.4 Display contents removal

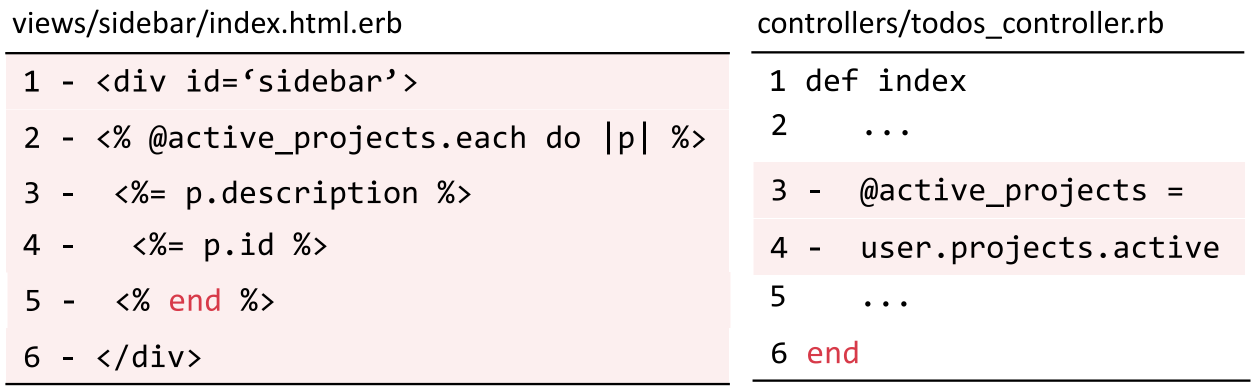
Obviously, one can remove an HTML tag to speed up the page loading. This strategy is indeed used in practice, as the `Tracks` example discussed in Section **??**. Whether an HTML tag is worthwhile to display cannot be determined automatically. Instead, what PowerStation can do is to make the removal easy and error-free, so that developers can easily try out different design options and eventually make an informed decision.

Identifying opportunities

Removing an HTML tag e does not guarantee to save page-loading time, because if the expensive computation needed by e is also needed by other HTML tags, removing e alone will not help performance much. The current prototype of PowerStation only suggests removing an HTML tag e if its contributing query that is not fed to any other HTML tag. This way, removing e can guarantee to save some data-processing time. Of course, future work can relax this checking criterion.

Generating patches

Removing an HTML tag e from the web page again involves changes to both the view component and the controller component of a web application. On the view side, PowerStation



views/sidebar/index.html.erb

```
- <div id=‘sidebar’>
- <% @active_projects.each do |p| %>
-  <%= p.description %>
-   <%= p.id %>
-  <% end %>
- </div>
```

controllers/todos_controller.rb

```
  def index
      ...
-     @active_projects =
-     user.projects.active
      ...
  end
```

(a) (b)

Figure 4.6: code change for removing

simply removes the specific HTML tag. On the controller side, PowerStation again analyzes control-dependency and data-dependency graph to remove code that was used only to help generate e.

To do so, PowerStation first identifies all the nodes in ADG that are associated with e's ID. PowerStation deletes those nodes, removes all condition checking whose two branches now execute exactly the same code because of those node deletions, and then check if there are any other ADG nodes that become useless and should be deleted — a node is useless if it has no out-going data-dependency or control-dependency edges. PowerStation repeats this process for several rounds until no more nodes are identified as useless.

We use the view file code snippet in Figure 4.6a as an example. The HTML tag shown here corresponds to the sidebar that lists all projects in the Tracks example discussed in Figure ??. Given this tag, PowerStation first identifies the Ruby expression `@active_projects`, and then checks the ADG to see how `@active_projects` is computed in the controller (Figure 4.6b). PowerStation also finds out that the `@active_projects` computed in Figure 4.6b is not used in anywhere else. Consequently, the content-removal change will simply delete the sidebar tag in the view file and the corresponding computation in the controller file, as shown in Figure 4.6.

4.5 The PowerStation Interface

PowerStation comes with a new interface to present the cost estimation and optimization information, and help developers explore different web-page design options. We discuss the PowerStation interface in this section.

4.5.1 Information display in browser

Showing view-centric cost estimation information. PowerStation visualizes the performance information obtained by dynamic profiling or static estimation (Section 4.3) through a heat-map with the more costly HTML element having a more red-ish background, as illustrated in Figure 4.7.

To generate this heat map, PowerStation reads the output of its view-centric cost estimation (Section 4.3) and creates a JavaScript file `interactive.js` that sets the background color of every HTML tag through "`$(tag-id).css(''background-color'', color);`", where `tag-id` is the unique HTML tag ID and `color` is computed based on the cost estimation for this HTML tag (Section 4.3). Web developers can choose to see different heat-maps with buttons on the web page, like "Real-time" (dynamic profiling results) and "Relative" (statically estimated results) in Figure 4.7. We set the color using the HSL color scheme, with more expensive tags rendered with smaller hue values (i.e., more red-ish) and cheaper tags with larger hue values (i.e., more blue-ish).

Showing view-aware optimization suggestions. `interactive.js` described above helps display not only data-processing cost but also alternative view-design options for various HTML tags. Users simply right click an HTML tag in the browser to get a list of design options, as shown in Figure 4.7. The implementation is straight-forward, given the unique ID of every HTML tag and the performance-enhancing opportunities identified through PowerStation static analysis as discussed in Section 4.4.

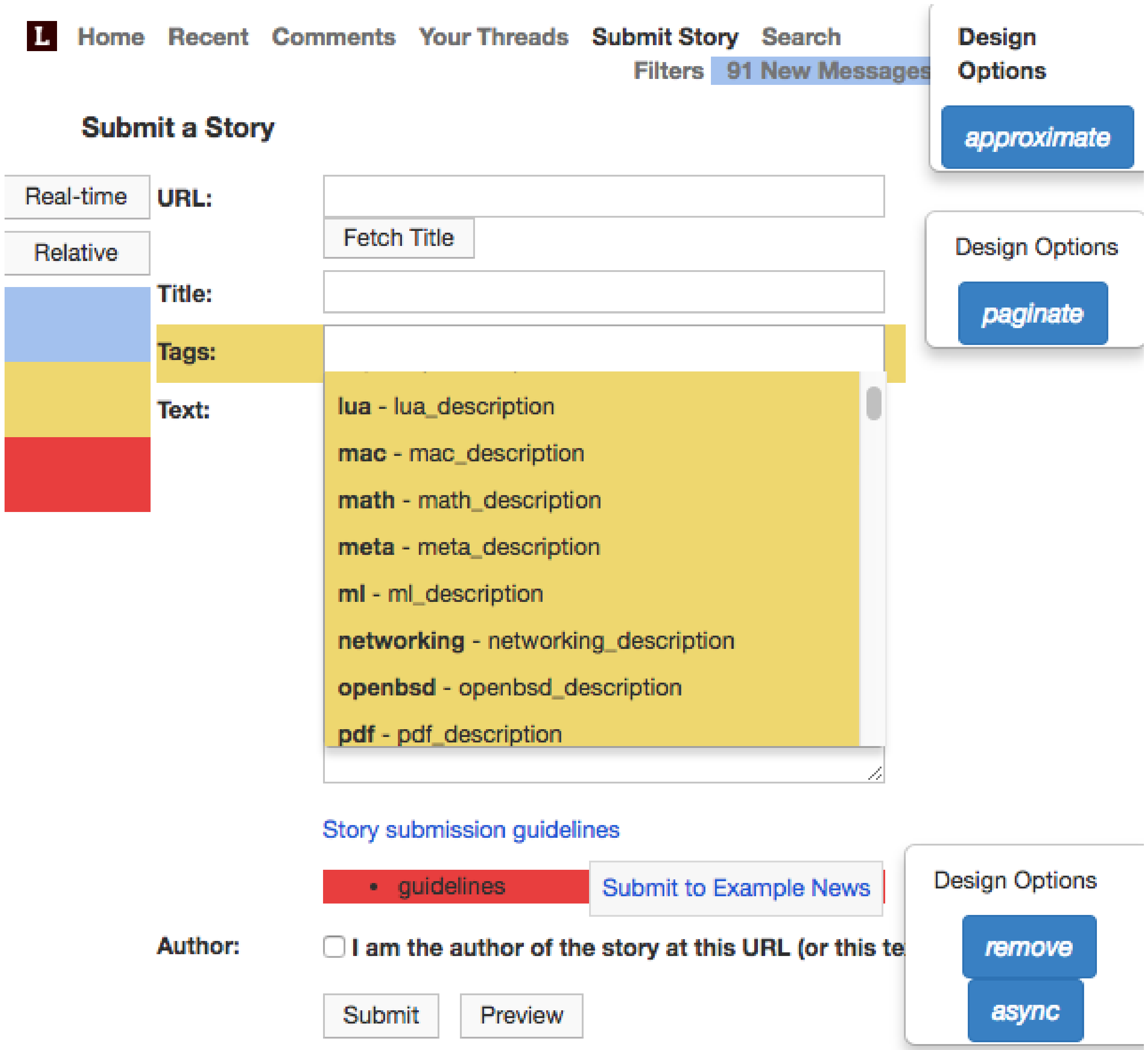


Figure 4.7: An example of PowerStation browser interface

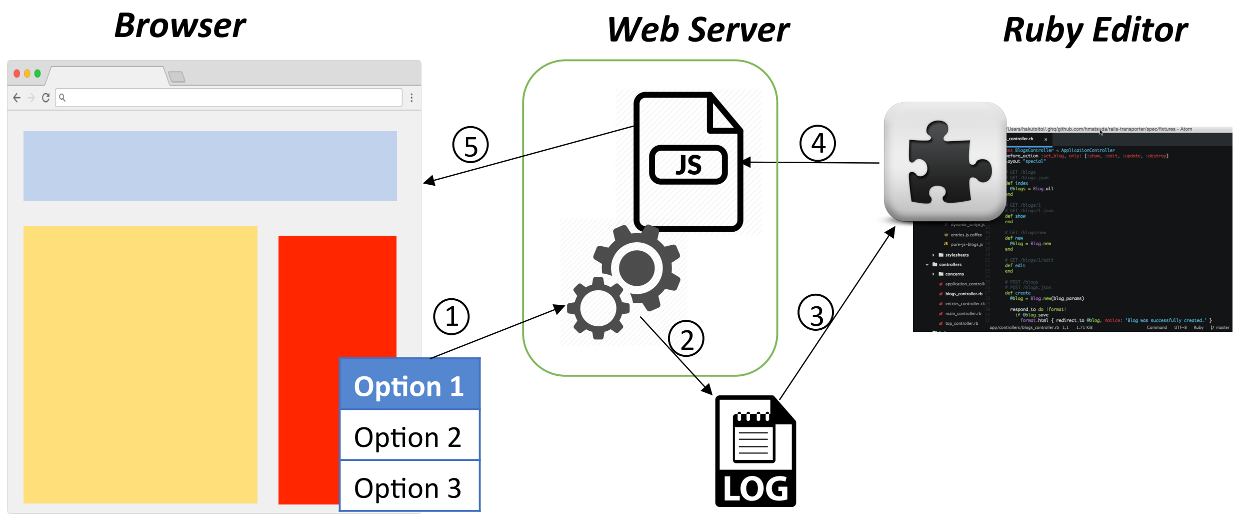


Figure 4.8: PowerStation interface implementation

4.5.2 Design-space exploration in browser

To help developers explore different performance–functionality design trade-offs, PowerStation further connects the browser-side information display and the Ruby editor side refactoring together: (1) developers first understand the data-processing cost of various HTML tags in the browser; (2) once developers choose an alternative design option for an HTML tag, the corresponding code refactoring will be automatically applied and displayed in the accompanying Ruby editor for developers to review; (3) once the source code is updated, the heat-map in the browser is updated accordingly. Developers can explore different design options, and eventually pick the best ones that suit their need.

To support this interface, PowerStation carefully uses JavaScript, IDE plugin, and other mechanisms to help the communication between the browser and the Ruby editor, as illustrated in Figure 4.8.[1]

First, PowerStation automatically instruments the web application under development to help communicate developers' design choices to the Ruby editor. Specifically, PowerStation

1. The current prototype of PowerStation assumes that the web-application under testing is deployed on the same machine as the Ruby editor.

adds a controller `_PANO_handle_request` into the web application. Whenever developers click a design-option button, like one of those blue *paginate*, *async*, *approximate*, *remove* buttons in Figure 4.7, PowerStation will send an HTTP request to invoke the `_PANO_handle_request` controller action (① in Figure 4.8), which then records the design-option type and the corresponding HTML tag ID into a web-server side file `request.log` (② in Figure 4.8).

In the editor, which we use RubyMine [?], PowerStation uses a thread to monitor the `request.log` file. Whenever this file is changed, this monitoring thread will trigger the plugin to apply corresponding code refactoring in the IDE, with all the code changes generated using algorithms described in Section 4.4 (③ in Figure 4.8).

After the code change, the data-processing cost estimation will be updated automatically, which results in updates to a performance profile and corresponding updates to `interactive.js` with changed background-color settings (④ in Figure 4.8). The changes in `interactive.js` lead to an automated refresh in the browser with the updated heat-map display, as we use the Ruby `react-rails-hot-loader` to enable automated refresh at every change in the Ruby source code or heat-map display code (⑤ in Figure 4.8).

4.6 Evaluation

Our evaluation focuses on four research questions: **RQ1**: Can PowerStation identify view-aware optimization opportunities from latest versions of popular web applications? **RQ2**: How much performance benefits can view-aware optimization provide? **RQ3**: Is the performance-functionality trade-off space exposed by PowerStation worthwhile for developers to explore? **RQ4**: Does PowerStation estimator estimate the per-tag data-processing cost accurately?

4.6.1 Methodology

Applications. We evaluate PowerStation using a suite of 12 open-source Ruby on Rails applications, including top 2 most popular Ruby applications from 6 major categories of

Table 4.1: Opportunities detected by PowerStation in 12 apps

App	Ds	Lo	Gi	Re	Sp	Ro	Fu	Tr	Da	On	FF	OS	SUM
pagi.	1	6	1	10	2	20	5	9	1	6	3	3	69
approx.	1	1	0	7	0	5	1	3	0	23	0	0	43
removal	1	2	0	7	0	4	1	2	2	2	0	0	22
asynch	1	2	0	2	0	2	1	2	2	2	0	0	15
SUM	4	11	1	26	2	31	8	16	5	33	3	3	149

Table 4.2: Speed up of 15 view changes

	Pagination						Asynchronous		Approximatio		
ID	Ro1	Tr1	Fu1	Re2	On1	Re1	Lo2	On5	Re2	On2	Tr2
Server Time Speedup (X)	19.4	13.5	6.8	4.7	2.1	1.8	37.8	1.1	2.1	1.4	1.2
End-to-end Time Speedup (X)	9.4	9.2	5.9	3.6	2.7	1.6	17.2	1.2	1.6	1.3	1.2

Every case is denoted by <application-short-name>-ID

web applications on GitHub: Discourse (Ds) and Lobster (Lo) are forums; Gitlab (Gi) and Redmine (Re) are collaboration applications; Spree (Sp) and Ror_ecommerce (Ro) are E-commerce 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. They have all been actively developed for years, with hundreds to tens of hundreds of code commits.

Workload. Since we cannot obtain real-world user data, we use synthetic data generation scripts released by previous work that to populate the databases following real-world data distribution and statistics. Similar to [100], we use the number of records in a web application's main database table to describe the workload size. By default, we use a 20,000-record workload unless otherwise specified. To our best knowledge, **all** the database sizes used in our evaluation are similar or smaller than the sizes in real-world web applications.

Table 4.3: Database sizes and page load time of 12 user-study cases

	Pagination			Asynchronous			Approxi	
ID	Re1	Ro1	Tr4	Re4	Tr3	Lo1	Re2	Tr1
DB Size (k-record)	2	0.8	2	2	2	20	100	100
Base Page Load Time (s)	2	1.9	2.5	2	1.9	1.8	2.5	2.5
New Page Load Time (s)	0.5	0.4	1	0.5	0.4	0.3	1	1

3 red IDs are cases from existing issue-tracking systems; the other 9 cases are all in latest versions discovered by PowerStation.
Every case is denoted in the same way as Table 4.2, with 6 common cases.

Platform. We profile the Rails applications on AWS Cloud9 platform [5], which has 2.5GB RAM and a 8-core CPU.

4.6.2 RQ1: how many opportunities does PowerStation identify?

As shown in Table 4.1, PowerStation can indeed identify many view-aware optimization opportunities. Specifically, PowerStation static analysis identifies 140 performance-enhancing opportunities from the current versions of our benchmark applications. Every type of optimization opportunities is identified from at least 8 applications.

These 149 opportunities apply to 119 unique HTML tags. For 101 HTML tags, only one view-change suggestion is made. For the remaining 18, PowerStation suggests two or three changes. Particularly, there are 15 HTML tags where *removal* and *asynchronous* loading both apply. Overall, these four types well complement each other.

4.6.3 RQ2: how much performance benefits?

To quantitatively measure the performance benefits of these alternative view designs, we randomly sampled 15 optimization opportunities identified above, with 6, 2, 4, and 3 cases from Pagination, Asynchronous (loading), Approximation, and Content Removal respectively, in 6 different applications. For each application, before and after optimization, we run a Chrome-based crawler that visits links randomly for 2 hours and measure the average end-to-end-latency and server-cost of every action. We then compute speedup accordingly.

As shown in Table 4.2, the performance benefits of these view changes are significant. By changing only one HTML tag, these 15 cases on average achieve 8.6× speed up on the server side and 4.5× speed up for end-to-end page load time. Among the four optimization types, *pagination*, *asynchronous* loading, and content *removal* have cases where the end-to-end page load time achieves about or more than 10× speedup.

4.6.4 RQ3: are alternate view designs worthwhile?

We evaluate the quality of a web page from two aspects: (1) how much users like the performance and functionality of a web page; (2) how much resources are needed to generate the page on the server side.

All four types of view changes suggested by PowerStation can help save server resources — *pagination*, *approximation*, and content *removal* all reduce tasks that need to be done by web and database servers; *asynchronous* loading provides more scheduling flexibility to servers.

Therefore, we believe an alternative web design is worthwhile for developers to explore, as long as users feel pages under this new design is *not worse* than the original one. To evaluate this, we conduct a thorough user study.

User study set-up

We recruited 100 participants on Amazon Mechanical Turk (Mturk). These participants are all more than 18 years old and living in the United States, with more than 95% MTurk Task Approval rate.

Our benchmark suite includes 12 web pages from 5 web applications. For each of these 12 baseline pages, PowerStation automatically generates a new page with exactly one HTML tag changed. We refer to the original page as *Base* and the one optimized by PowerStation as *New*. These 12 web pages cover all four types of view changes, with exactly 3 cases in each type. Furthermore, for every change type[2], we cover one case from on-line issue reports — these changes were already adopted by developers to fix performance problems in previous versions of web applications, and some cases discovered by PowerStation in current versions of these applications. We also reuse cases from Table 4.2 as much as we can.

Since the performance advantage of *New* pages depends on the database size, to ease comparison, we populate the database for each benchmark so that the load-time difference between the *Base* version and the *New* version is exactly 1.5 seconds. The detail settings are shown in Table 4.3.

Each participant is assigned 8 tasks. In each task, they are asked to click two links one by one, and then answer questions about (1) which page they think is faster ("Performance" in Table 4.4); (2) which page they think delivers more or better organized content ("Functionality" in Table 4.4); and (3) which page do they like more with everything considered ("Overall" in Table 4.4). These two links are the *Base* and *New* versions of one benchmark, with random ordering between them.

User study results

A summary of the user study results is shown in Table 4.4, and the questionnaire and raw data are available on Panorama webpage [33]. In this table, we show the percentage of users who think New is better minus those who think Base is better, which we refer to as the *net* benefit of the new design, for every type of refactoring and every question (Performance, Functionality, and Overall). Users are given the two pages in random order, and are not aware of the view-design difference between the two pages in advance.

2. Except for approximation, as we did not find performance issue reports in these 12 web applications that are solved by approximation.

A short answer to our research question is “Yes”. In fact, for all type of view-changing optimization, users think the New design is *not worse* than the Base design. Particularly, for *asynchronous* loading, *pagination*, and *removal*, the new designs clearly win more users than the baseline designs.

In terms of performance, the net win for the New design is clear. Many users indeed notice the 1.5-second difference in the page-load time. In 10 out of 12 cases, the New design has a net positive benefit on more than 30% of the participants.

In terms of functionality, the results are quite interesting. For *approximation* and *pagination*, many users did notice the content difference, leading to the Base design winning about 10% of users. However, for *asynchronous* loading and content *removal*, surprisingly, many users neither notice the content difference nor think New design delivers worse contents. It could be that removing contents made the page cleaner to some users. For example, after removing the sidebar in Tracks (Tr5 in Table 4.3), some participants like it because “Adding the sidebar makes scrolling harder.”

In terms of overall perception, the New design has a **net win**. Among the four types of optimizations, paginations and asynchronous loading are the most appealing to users, while approximation is the least appealing.

We also conducted another set of user study with another 100 participants, where we use an even larger (2-10×) database size and hence make the page-load time differences between New design and Base design even bigger (3 seconds). We do observe that more participants noticed the performance advantage of the New design. However, we also observe that the overall perception only goes up a little bit more for the New design. We skip the details for the space constraints.

Table 4.4: Net user perception enhancement by New design
(% users who prefer New − % users who prefer Base)

	Approximation	Asynch	Paginate	Removal
Performance	23.50%	31.00%	45.00%	35.50%
Functionality	-7.50%	17.50%	-13.00%	5.50%
Overall	1.00%	18.50%	28.00%	10.50%

4.6.5 RQ4: how accurate is the PowerStation estimator?

We use the web page shown in Figure 4.7 as a case study. 15 HTML tags on this page render dynamically generated contents. With 200 database records, dynamic profiling shows that the story tag is the top performance bottleneck, followed by the guideline tag and then the message-count the cheapest. When the workload increases to 2000 and 20000 records, dynamic profiling shows that the guideline tag is the top bottleneck, followed by the story tag. These results match the performance gains we can get by optimizing these three tags: using 20000 records, asynchronously loading or removing the guideline text can reduce the page end-to-end load time by more than 1 second; paginating the story-tag can speed up the page load time by about 100 milliseconds; approximating the message count does not change page load time.

The static mode of PowerStation estimator can indeed predict performance bottleneck without running the application — the guideline text gets the highest complexity score (5) among all 15 tags, followed by the story tag (4), and then the message-count tag (3), with the remaining 12 tags getting 0 points.

4.6.6 Threats to validity

Threats to the validity of our work could come from multiple sources. **Internal Validity**: The HTML tags that are dynamically generated through JavaScript can not be detected or analyzed by PowerStation. **External Validity**: The 12 applications in our benchmark suite may not represent all real-world applications; The synthesized databases may not represent real-world workloads; The machine and network settings of our profiling may differ from real users' setting; 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.

4.7 Conclusion

It is increasingly challenging to develop web applications that can deliver both good functionality and desired performance. We present PowerStation, a tool that helps web developers explore the performance-functionality trade-off space in their web application design. The PowerStation estimator provides developers with data-processing cost information for every HTML tag that renders dynamically generated data, while the PowerStation optimizer identifies and automates view-changing refactoring that can greatly improve performance. The PowerStation interface integrates estimator and optimizer together to enable effective web application design.

CHAPTER 5

MANAGING DATA CONSTRAINTS IN DATABASE-BACKED WEB APPLICATIONS

In this chapter, we aim to answer four key research questions about real-world database-backed web applications, as listed in Table 5.1 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.

Table 5.1: 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 140 studied issues
What	51% of 140 studied issues
When	10% of 140 studied issues
How	18% of 140 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

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 5.2.

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 5.3.

For RQ3, we thoroughly investigated 140 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 5.4.

For RQ4, we developed tools that automatically identify many data-constraint problems in the latest versions of these 12 applications, as highlighted in Table 5.1. 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 5.5.

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 [26], a project that aims to solve database-related problems in ORM applications.

5.1 Methodology

5.1.1 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

Table 5.2: # 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

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.

5.1.2 Issue selection

Section 5.4 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 5.2.

Table 5.3: # 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.

5.2 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 5.1. 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 **??** 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.

5.2.1 How many constraints are there?

As shown in Table 5.3, 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.

5.2.2 Where are the constraints?

As shown in Table 5.3, 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 **??**). Furthermore, inconsistencies between them can lead to application crashes as in the example shown in Figure **??**. This led to the next few study items.

What DB constraints are missing in applications? Table 5.4 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 **??**. In fact, we found many real-world issues reporting this problem (Sec. 5.4.1), 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

Table 5.4: # 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 5.5: # 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%)

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.

False-positive analysis Besides the "Others" row in Table 5.4, 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 5.5 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 [74].

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.

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

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 [36, 35]: 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.

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 5.5 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 5.5 (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

Table 5.6: 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%)	-	-

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.

5.2.3 What types of constraints are there?

Standard types Table 5.6 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` 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.

5.3 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 5.7, 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.

Table 5.7: 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 **??** except for custom sanity checks are considered.
Red apps use a release as a version; Black apps use every 100 commits as a version.

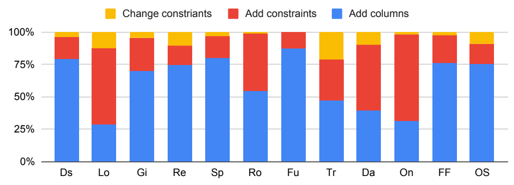


Figure 5.1: Breakdown of # of adding/changing constraints

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 5.1) 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.

5.4 Constraint-related issues

We categorize 140 real-world issues into 4 types as shown in Table 5.8.

5.4.1 WHERE is the constraint specified?

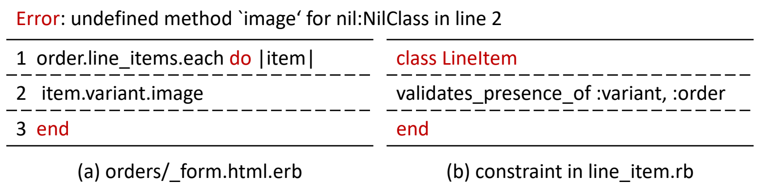
As discussed, application and DB constraints for the same data field can be inconsistent with each other. Such inconsistencies contributed to 24 out of the 114 issues.

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 **??** 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 [44], an on-line shopping system,

Table 5.8: 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 5.2: Constraint mismatch in Spree

has 4 issues caused by administrators modifying database content through direct SQL requests. Discourse [13] even has scripts that bypass model constraints to import other forum applications' data.

Figure 5.2 shows such an issue [45] 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 5.2a.

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 5.2.2), 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 5.5.

5.4.2 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 140 issues.

Conflict with user needs

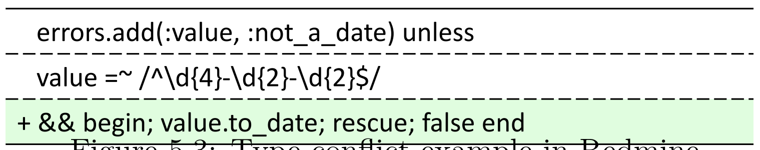
Users sometimes would relax an existing constraint, such as increasing the input length of name field in tracker from 30 to 100 (Redmine-23235 [40]). These contribute to about 10% of the issues in our study. Developers usually satisfy the users' desires and change constraints accordingly.

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.

Conflict with application needs

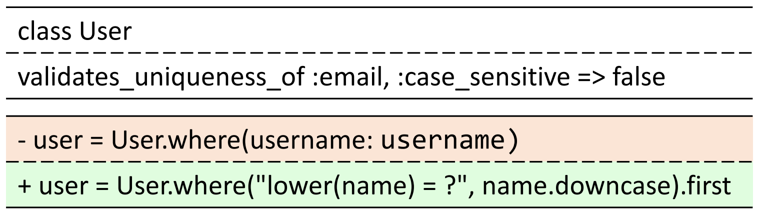
Many constraints are created to guarantee program invariants that are crucial to applications' functional correctness. Constraints that are insufficient or even conflicting with how the corresponding data is used by the application contribute to more than one third of all the issues in our study.

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 [41], 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"



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

Figure 5.3: 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 5.4: Case sensitivity conflict example in Gitlab

to check whether the input can really be converted to a date or not in the custom validate function shown in Figure 5.3.

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 [22], 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 5.4.

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[46], 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. 5.5.

5.4.3 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.

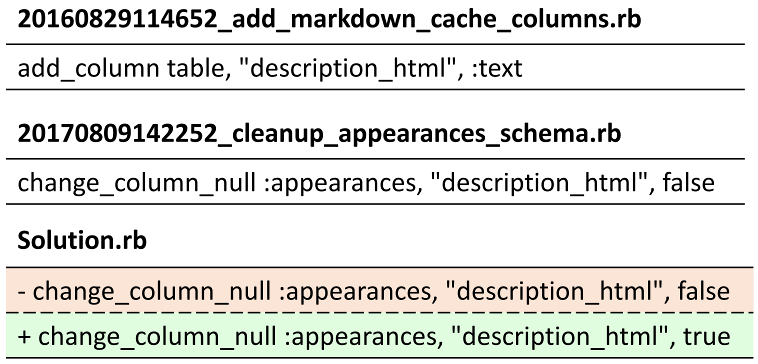
Read path When a constraint is newly created or tightened along a DB-record loading code path (e.g., front-end constraint or application sanity-check changes), an incompatible new constraint can cause failures in loading old data and hence severe functionality problems. The Diaspora example in Section **??** 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 [23] 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 5.5, the `decription_html` column was added to Gitlab before version 9.4.5 (in the "20160829..." migration file) and was filled with `null`s by default.[2] Later on, in the "20170809..." migration (shown in the Figure 5.5), 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 5.5.

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 [15], a user complained that she made a small edit to an old post's title, but was unable to save with an error 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.

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 5.5: Old data conflicts with new constraints in Gitlab

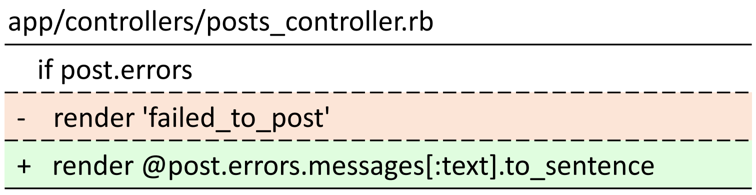
Summary Given the frequent constraint addition and changing in web applications, it is inevitable that old data may become incompatible with new constraints. It would be helpful if automated tools can provide warnings for developers when constraints become tighter in a new version, particularly (1) if 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 5.5.

5.4.4 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 some-

3. Section 5.5 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 5.6: Unclear error message in Diaspora

times override the default message with uninformative generic messages (12 cases), which led to user complaints. For example, in a Diaspora issue [12], 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 5.6.

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 5.5.

5.5 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.

5.5.1 Where issues

As discussed in Section 5.2.2, 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 5.9, 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.

Table 5.9: # 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

5.5.2 What issues

We built a checker to detect "case-sensitivity conflicts" discussed in Section 5.4.2. 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 [101]; 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.

5.5.3 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 5.3) to see if new constraints are added or existing constraints are tightened. We then further check whether

Table 5.10: 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 5.11: 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%

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 problematic upgrade pointed out by the 12 known issues in our study (Section 5.4.3), and confirmed that it can detect all these 12 issues with no false positives. It did not find problems with the latest upgrade of these applications.

5.5.4 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.

We found that 5 APIs' default messages can be more informative, as shown in Table 5.10.

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 5.3, 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.

5.6 Discussion

5.6.1 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.

5.6.2 Threats to Validity

Internal Threats to Validity: As discussed in Section **??** and 5.2, 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 5.2.2 and 5.5.

External Threats to Validity: The 12 applications in our study clearly may not represent all real-world applications; the 140 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 **??**, 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 [17]	1	2	3	3	1	10
zulip [48]	1	4	2	0	0	7
redash [37]	0	2	0	0	3	5

5.7 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.

CHAPTER 6
RELATED WORK

6.1 Performance

Empirical studies Previous work confirmed that performance bugs are prevalent in open-source C/Java programs and often take developers longer time to fix than other types of bugs [68, 104]. Prior work [84] studied the performance issues in JavaScript projects. We target performance problems in ORM applications that are mostly related to how application logic interacts with underlying database and are very different from those in general purpose applications. Our recent work [95] looked into the database performance of ORM applications and discussed how better database optimization and query translation can improve ORM application performance. No issue report study or thorough profiling was done. In contrast, our paper performs a comprehensive study on all types of performance issues reported by developers and discovered using profiling. Unnecessary data retrieval (UD), content display trade-offs (DT), and part of the inefficient data accessing (ID) anti-patterns are the only overlap between this study and our previous work [95] .

Inefficiencies in ORM applications Previous work has addressed specific performance problems in ORM applications, such as locating unneeded column data retrieval [56], N+1 query [55], pushing computation to the DBMS [58], and query batching [57, 83, 93]. While effective, these tools do not touch on many anti-patterns discussed in our work, like unnecessary computation (UC), inefficient rendering (IR), database designs (MF, MI), functionality trade-offs (FT), and also do not completely address anti-patterns like inefficient computation (IC) and inefficient data accessing (ID).

Performance issues in other types of software Much research was done to detect and fix performance problems in general purpose software [68, 78, 77, 81, 62, 91, 92]. Detecting and fixing ORM performance anti-patterns require a completely different set of techniques that understand ORM and underlying database queries.

6.2 Correctness

Verifying data constraints. Prior work has investigated verifying database-related constraints. ADSL [52] 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 [85, 90] check whether a set of DB constraints still hold when DB schema evolves while Caruccio [53] conducts a survey of related work in this domain. Pan [82] 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[60, 88, 59], DB applications [89], synthesizing a new DB program with a new scheme given the original program with an old scheme [90], and handling chains of interactive actions [61].

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 [70, 76, 97, 50]. 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, Wander [71] 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 [94] adds constraints as extra assumptions to help synthesize better query plans, and Quro [93] 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 [100, 96, 55, 56] and client-side performance problems [84] but not data-constraint problems.

CHAPTER 7
FUTURE WORK

My thesis has focused on web applications with SQL-based database backend. I believe similar cross-stack performance and correctness problems widely exist in general big-data software — any software maintains and processes a large amount of data, which are stored in either SQL or non-SQL data stores under various setting including centralized computing, high performance computing, or high performance computing. My future research goal is to continue to improve the performance and correctness of big-data software through different stages of software development: design, coding, testing, and maintenance. I believe achieving this requires better interactions between developers and software, which involves inter-disciplinary solutions: software engineering, programming language, database, etc.

View-Driven application design framework. Panorama is the first attempt to apply view-driven optimization by synthesizing efficient web interface designs for developers facing performance and functionality trade-offs. However the view-design optimizations provided and the interactions between developers are still limited. I believe a view-driven application design framework utilizing developers' domain-specific knowledge is necessary to construct efficient applications from the design stage. The framework will enable developers to drag and drop the view components and assign priority label to the components. The priority label is to indicate those that should be rendered first. The application logic behind the view component could be directly written by the developers or even synthesized by the framework through input/output examples provided by the developers. Given the priority and resource budget (e.g. memory), the framework will automatically analyze the code to devise a plan to render each of the pages in the application, with the goal to reduce the time of the high priority elements as much as possible by incorporating existing optimizations we have explored. At the same time, the framework should be open to new optimization techniques by exposing the analysis framework for easy integration. I believe this framework will benefit any user-facing applications where data processing is performance bottleneck.

Improving performance using data constraints. While studying the data constraints for the software correctness, I found that constraints could also be used to improve the performance. For example, an inclusion constraint which ensures the values of a field are included in a given set indicates that the field can be stored as an ENUM type in the database which both saves the space and speeds up the query time. I will study the real-world applications' source code to extract more patterns of inferring performance optimization from data constraints. Based on the patterns, I will build static analysis tools to systematically detect the optimization opportunities using the existing data constraints.

Fixing application code and database schema inconsistency. Applications with database backends often undergo schema evolution frequently during their life cycle. The corresponding application code should evolve according to the schema change, which requires non-trivial efforts and is error-prone. I will firstly conduct an empirical study to evaluate the prevalence and severity of the problems caused by application code and database schema inconsistency in real-world web applications. I then will build tool support to detect and fix the inconsistency through cross-stack program analysis.

Program synthesis for big-data analytics. Data analytics becomes increasingly important in big data applications. While the ultimate goal is to subset large amounts of data to extract insights, a study [10] claims that a typical data scientist spends 50-80% of their time performing so-called "janitor work" of data science, which includes tasks such as data reshaping, cleaning, and imputing (i.e., filling missing entries). Previous work tries to solve this problem [87] by synthesizing the data preparation from input/output example, however, this approach requires the output example to be partially evaluated from the input data, which still requires non-trivial efforts and often are not-feasible. Also, it does not scale well on large input files. I believe there is still a long way to go to boost the productivity of data scientists by program synthesis. I plan to apply my experience in synthesizing complex data pipelines from Juypter notebooks by target to provide support for big-data analytics through different stages including data preparation, data visualization, etc.

Improving the Correctness of Cloud Computing Systems. Cloud computing systems and web applications share some properties. For example, they all need to manage large amount of data and the communication among different components could result in data maintenance problems. At the same time, distributed systems consist of more components, involve in more channel for data generation including node message and persistent file, and often go through rolling upgrade with nodes of multiple versions communicating altogether, which makes data maintenance in cloud computing systems harder and subtler. I believe the essence of data maintenance bugs is the same in distributed systems and in web applications. I will employ my experience in detecting and fixing data maintenance problems in web applications to improve the correctness of cloud computing systems.

REFERENCES

[1] *Active Support Instrumentation.* http://guides.rubyonrails.org/active$_{s}upport_{i}nstrumentation.html/$.

[2] *Amazon.* An online e-commerce application. https://amazon.com/.

[3] asynchronous rendering library. https://github.com/renderedtext/render-async.

[4] *AutoAdmin.* For database systems self-tuning and self-administering. https://www.microsoft.com/en-us/research/project/autoadmin/.

[5] Aws cloud9. https://aws.amazon.com/cloud9/.

[6] *AWS instance types.* https://aws.amazon.com/tw/ec2/instance-types/.

[7] *Browser Ranking.* http://www.zdnet.com/article/chrome-is-the-most-popular-web-browser-of-all/.

[8] *Bullet.* A library used to solve N + 1 query problem for Ruby on Rails. https://github.com/flyerhzm/bullet/.

[9] chrome development tool. https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/reference.

[10] *Data janitor.* https://nyti.ms/3nvyObD.

[11] *Diaspora.* A social-network application. https://github.com/diaspora/diaspora/.

[12] *Diaspora-5090.* https://github.com/diaspora/diaspora/issues/5090.

[13] *Discourse*. A blog application.
https://github.com/discourse/discourse/.

[14] Discourse-4663 issue report.
https://github.com/discourse/discourse/pull/4663.

[15] *Discourse-89148*. https://meta.discourse.org/t/89148.

[16] *Django*. https://www.djangoproject.com/.

[17] *Django-cms*. An enterprise content management system.
https://github.com/divio/django-cms/.

[18] *Download PowerStation*. https://bit.ly/2NYFRs3.

[19] *Find your new favorite web framework*.
https://hotframeworks.com/.

[20] *Github*. https://github.com/.

[21] *Gitlab*. A software to collaborate on code.
https://github.com/gitlabhq/gitlabhq/.

[22] *Gitlab-24493*. https://gitlab.com/gitlab-org/gitlab-ce/issues/24493.

[23] *Gitlab-36919*. https://gitlab.com/gitlab-org/gitlab-ce/issues/36919.

[24] *Hibernate*. http://hibernate.org/.

[25] *Hyperloop*. http://hyperloop.cs.uchicago.edu.

[26] *Hyperloop*. https://hyperloop-rails.github.io/vibranium/.

[27] jquery. https://jquery.com/.

[28] *JRuby*. https://github.com/jruby/jruby.

[29] *Lobsters*. A forum application.
https://www.github.com/jcs/lobsters/.

[30] *N + 1 query problem*. https://www.sitepoint.com/silver-bullet-n1-problem/.

[31] *OpenStreetMap*. A map service application.
https://github.com/openstreetmap/openstreetmap-website/.

[32] *Pagination*. A library used in webpage displaying.
https://github.com/mislav/will$_{paginate}$/.

[33] Panorama website. https://hyperloop-rails.github.io/panorama/.

[34] *PowerStation*. www.hyperloop.cs.uchicago.edu/powerstation/.

[35] *Rails Uniqueness API*. https://github.com/rails/rails/blob/master/activerecord/lib/active_record/va

[36] *Rails Uniqueness Problem*. https://thoughtbot.com/blog/the-perils-of-uniqueness-validations.

[37] *Redash*. An application to connect your company's data.
https://github.com/getredash/redash/.

[38] *redmine*. https://redmine.org/.

[39] *Redmine-23334*. https://redmine.org/issues/23334/.

[40] *Redmine-25235*. http://www.redmine.org/issues/25235/.

[41] *Redmine-9394*. http://www.redmine.org/issues/9394/.

[42] *Ruby on Rails*. http://rubyonrails.org/.

[43] *RubyMine*. https://www.jetbrains.com/ruby/.

[44] *Spree*. A ecommerce application.
https://github.com/spree/spree/.

[45] *Spree-3829*. https://github.com/spree/spree/issues/3829.

[46] *Spree-6673*. https://github.com/spree/spree/issues/6673.

[47] will-paginate tool package.
https://github.com/mislav/will-paginate.

[48] *Zulip*. A powerful team chat system.
https://github.com/zulip/zulip/.

[49] Akamai and Gomez.com. *How Loading Time Affects Your Bottom Line.*
https://blog.kissmetrics.com/loading-time/.

[50] Muath Alkhalaf, Shauvik Roy Choudhary, Mattia Fazzini, Tevfik Bultan, Alessandro Orso, and Christopher Kruegel. 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*, pages 56–66. ACM, 2012.

[51] Michael Armbrust, Eric Liang, Tim Kraska, Armando Fox, Michael J. Franklin, and David A. Patterson. Generalized scale independence through incremental precomputation. In *SIGMOD*, pages 625–636, 2013.

[52] Ivan Bocić, Tevfik Bultan, and Nicolás Rosner. Inductive verification of data model invariants in web applications using first-order logic. *Automated Software Engineering*, 26(2):379–416, 2019.

[53] Loredana Caruccio, Giuseppe Polese, and Genoveffa Tortora. Synchronization of queries and views upon schema evolutions: A survey. *ACM Transactions on Database Systems (TODS)*, 41(2):9, 2016.

[54] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *ICSE*, 2014.

[55] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *ICSE*, pages 1001–1012, 2014.

[56] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. In *ICSE*, pages 1148–1161, 2016.

[57] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). pages 931–942.

[58] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *PLDI*, pages 3–14, 2013.

[59] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *CIDR*, 2017.

[60] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. HoTTSQL: Proving query rewrites with univalent SQL semantics. In *PLDI*, pages 510–524, 2017.

[61] Alin Deutsch, Monica Marcus, Liying Sui, Victor Vianu, and Dayou Zhou. A verifier for interactive, data-driven web applications. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 539–550. ACM, 2005.

[62] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *FSE*, pages 59–70, 2008.

[63] Wenfei Fan, Floris Geerts, and Leonid Libkin. On scale independence for querying big data. In *PODS*, pages 51–62, 2014.

[64] Anne Farrell and Henry Hoffmann. Meantime: Achieving both minimal energy and timeliness with approximate computing. In *USENIX*, pages 421–435, 2016.

[65] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[66] Michael Furr, Jong-hoon David An, Jeffrey S Foster, and Michael Hicks. Static type inference for ruby. In *SAC*, 2009.

[67] Paul Graham. *Startup = Growth.* http://paulgraham.com/growth.html.

[68] Jin Guoliang, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *PLDI*, pages 77–88, 2012.

[69] Yannis E Ioannidis and Viswanath Poosala. Histogram-based approximation of set-valued query-answers. In *VLDB*, pages 174–185, 1999.

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

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

[72] Yu Lin, Cosmin Radoi, and Danny Dig. Retrofitting concurrency for android applications through refactoring. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 341–352. ACM, 2014.

[73] Greg Linden. *Marissa Mayer at Web 2.0.* http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html/.

[74] Guy Lohman. *Is Query Optimization a "Solved" Problem?* https://wp.sigmod.org/?p=1075.

[75] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, pages 245–256, 2003.

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

[77] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. CARAMEL: detecting and fixing performance problems that have non-intrusive fixes. In *ICSE*, pages 902–912, 2015.

[78] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: detecting performance problems via similar memory-access patterns. In *ICSE*, pages 562–571, 2013.

[79] Semih Okur, Danny Dig, Yu Lin, et al. Study and refactoring of android asynchronous programming. 2015.

[80] Semih Okur, David L Hartveld, Danny Dig, and Arie van Deursen. A study and toolkit for asynchronous programming in c. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1117–1127. ACM, 2014.

[81] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *PLDI*, pages 369–378, 2015.

[82] Kai Pan, Xintao Wu, and Tao Xie. Guided test generation for database applications via synthesized database interactions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(2):12, 2014.

[83] Karthik Ramachandra, Chavan Mahendra, Guravannavar Ravindra, and S Sudarshan. Program transformations for asynchronous and batched query submission. In *TKDE*, pages 531–544, 2015.

[84] Marija Selakovic and Michael Pradel. Performance issues and optimizations in javascript: an empirical study. In *ICSE*, pages 61–72, 2016.

[85] Rohit Singh, Vamsi Meduri, Ahmed Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. Generating concise entity matching rules. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1635–1638. ACM, 2017.

[86] Jeffrey D. Ullman and Jennifer Widom. *A First Course in Database Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[87] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 452–466, 2017.

[88] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. Speeding up symbolic reasoning for relational queries. *PACMPL*, 2(OOPSLA):157:1–157:25, 2018.

[89] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. Verifying equivalence of database-driven applications. In *Proceedings of the ACM on Programming Languages*, pages 56:1–56:29, 2017.

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

[91] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *PLDI*, pages 419–430, 2009.

[92] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In *PLDI*, pages 174–186, 2010.

[93] Cong Yan and Alvin Cheung. Leveraging lock contention to improve oltp application performance. pages 444–455.

[94] Cong Yan and Alvin Cheung. Generating application-specific data layouts for in-memory databases. *Proc. VLDB Endow.*, pages 1513–1525, 2019.

[95] Cong Yan, Junwen Yang, Alvin Cheung, and Shan Lu. Understanding database performance inefficiencies in real-world web applications. In *CIKM*, 2017.

[96] Cong Yan, Junwen Yang, Alvin Cheung, and Shan Lu. Understanding database performance inefficiencies in real-world web applications. In *CIKM*, 2017.

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

[98] Junwen Yang, Utsav Sethi, Cong Yan, Alvin Cheung, and Shan Lu. Managing data constraints in database-backed web applications. In *Proceedings of the International Conference on Software Engineering*, 2020.

[99] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. How

not to structure your database-backed web applications: a study of performance bugs in the wild. In *ICSE*, 2018.

[100] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *ICSE*, pages 800–810, 2018.

[101] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. Powerstation: Automatically detecting and fixing inefficiencies of database-backed web applications in ide. In *26th Foundations of Software Engineering (FSE'18 Demostration Track)*, 2018.

[102] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018.

[103] Junwen Yang, Cong Yan, Chengcheng Wan, Shan Lu, and Alvin Cheung. View-centric performance optimization for database-backed web applications. In *IEEE/ACM 41th International Conference on Software Engineering (ICSE)*. IEEE, 2019.

[104] Zaman, Shahed, Bram Adams, and Ahmed E. Hassan. A qualitative study on performance bugs. In *MSR*, pages 199–208, 2012.