How *not* to structure your database-backed web applications: a study of performance bugs in the wild [*]

Junwen Yang
Pranav Subramaniam
Shan Lu
University of Chicago
{junwen, psubramaniam, shanlu}@uchicago.edu

Cong Yan
Alvin Cheung
University of Washington
{congy, akcheung}@cs.washington.edu

ABSTRACT

Many web applications use databases for persistent data storage, and using Object Relational Mapping (ORM) frameworks is a common way to develop such database-backed web applications. Unfortunately, developing efficient ORM applications is challenging, as the ORM framework hides the underlying database query generation and execution. This problem is becoming more severe as these applications need to process an increasingly large amount of persistent data. Recent research has targeted specific aspects of performance problems in ORM applications. However, there has not been any systematic study to identify common performance anti-patterns in real-world such applications, how they affect resulting application performance, and remedies for them.

In this paper, we try to answer these questions through a comprehensive study of 12 representative real-world ORM applications. We generalize 9 ORM performance anti-patterns from more than 200 performance issues that we obtain by studying their bug-tracking systems and profiling their latest versions. To prove our point, we manually fix 64 performance issues in their latest versions and obtain a median speedup of 2× (and up to 39× max) with fewer than 5 lines of code change in most cases. Many of the issues we found have been confirmed by developers, and we have implemented ways to identify other code fragments with similar issues as well.

CCS CONCEPTS

• **Software and its engineering → Software performance**;

KEYWORDS

performance anti-patterns, Object-Relational Mapping frameworks, database-backed applications, bug study

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 [32], 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 [24], while every extra 0.5 second of latency reduces the overall traffic by 20% [35].

To manage large amounts of data, modern web applications often follow a two-stack architecture, with a front-end application stack fulfilling application semantics and a back-end database management system (DBMS) storing persistent data and processing data retrieval requests. 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 [22], Django for Python [9], and Hibernate for Java [14]. 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.

Unlike performance problems on the client side, which have been well studied in prior work [34, 41], the cross-stack performance problems on the server side are under-studied, which unfortunately are the key to the data-processing efficiency of ORM applications. Although recent work [26, 27, 29, 46] has looked at specific performance problems in ORM applications, there has been no comprehensive study to understand the performance and scalability of real-world ORM applications, the variety of performance issues that prevail, and how such issues are addressed.

Given the above, we target three key research questions about real-world ORM applications in this paper:

[*] https://hyperloop.cs.uchicago.edu

ICSE'18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180194

- **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?

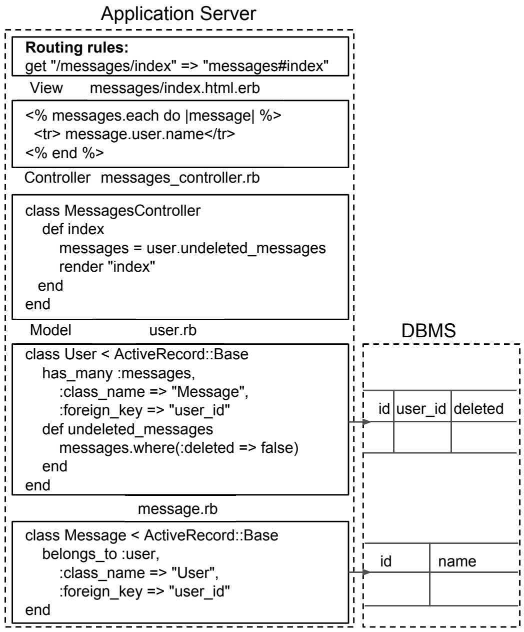
To answer these questions, we conduct a comprehensive two-pronged empirical study on a set of 12 Rails applications representing 6 common categories that exercise a wide range of functionalities provided by the ORM framework and DBMS. We choose Rails as it is a popular ORM framework [11]. We carefully 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 3), 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 4 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). 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 6). Our fixes achieve 2× median speedup (and up to 39 ×) 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 7).



```
get "/messages/index" => "messages#index"
```

messages/index.html.erb

```
<% messages.each do |message| %>
  <tr> message.user.name</tr>
<% end %>
```

messages_controller.rb

```
class MessagesController
   def index
      messages = user.undeleted_messages
      render "index"
   end
end
```

user.rb

```
class User < ActiveRecord::Base
   has_many :messages,
      :class_name => "Message",
      :foreign_key => "user_id"
   def undeleted_messages
      messages.where(:deleted => false)
   end
end
```

message.rb

```
class Message < ActiveRecord::Base
   belongs_to :user,
      :class_name => "User",
      :foreign_key => "user_id"
end
```

Figure 1: Structure of an example Rails application

Overall, our comprehensive study provides motivations and guidelines for future research to help avoid, detect, and fix cross-stack performance issues in ORM applications. We have prepared a detailed replication package for all the performance-issue study, profiling, and program analysis conducted in this paper. This package is available on the webpage of our Hyperloop project [16], a project that aims to solve database-related performance problems in ORM applications.

2 BACKGROUND

Our study focuses on applications written in Ruby on Rails (Rails). Ruby is among the top 3 popular languages on GitHub [38], and Rails is among the top 3 popular web application frameworks [11]. Many widely used applications are built upon Rails, such as hulu [15], gitlab [13], airbnb [2], etc. Compared to other popular ORM frameworks such as Django [9] and Hibernate [14], Rails has 2× more applications on github with 400 more stars than Django and Hibernate combined. As Rails provides similar functionalities as Django and Hibernate, we believe our findings can apply to applications built on top of those frameworks as well.

Like other applications built on top of an ORM framework, Rails applications are structured based on the model-view-controller (MVC) architecture. We illustrate this using an example shown in Figure 1, which is abridged from a forum application that allows users to publish posts and comments. First, developers design *model* classes that inherit from a special `ActiveRecord` super class, such as `User` and `Message` in Figure 1, where their corresponding fields are stored persistently in the DBMS. The associations between model classes, chosen from `has_many`, `has_one`, and `belongs_to`, need to be explicitly declared in *model* classes, such as the "`has_many :messages`" specified in the `User` class and the "`belongs_to :user`" specified in the `Message` class. After that, they design *controllers*, such as `MessagesController.rb` in Figure 1 that contains multiple `actions`, with each action determining

Table 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

how the application responds to a specific web-page request. Inside an action there is code to retrieve database data through queries transparently translated by the ORM. Finally, the retrieved data is rendered via *views* that are often written in a template language, as shown in `index.html.erb` in Figure 1. Such views determine how the retrieved data is displayed in a client's browser.

The life cycle of a Rails application, and ORM applications in general, is as follows. When receiving a client HTTP request like "`http://.../messages/index`", the application server first looks up the routing rules, shown at the top of Figure 1, to map this request to the `index` action inside `MessagesController`. When the `index` action executes, it invokes the `@user.undeleted_messages` function, which calls `messages. where(...)`. The call to the Rails API `where` is dynamically translated to a SQL query by the Rails framework to retrieve data from the DBMS. The query results are then serialized into model objects and stored in `@messages`. The `index` action then calls `render "index"` to render the retrieved data in `@messages` using the `index.html.erb` template.

3 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.

3.1 Application Selection

As mentioned in Section 2, we focus on Rails applications. Since it is impractical to study all open-source Rails applications (about 200 thousand of them on GitHub [12]), we focus on 6 popular application categories[1] as shown in Table 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 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 1 have been developed for 5 to 12 years. They are all in active use and range from 7K lines of code (Lobsters [17]) to 145K lines of code (Gitlab [13]).

Table 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 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

3.2 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 [13]) or similar application's website (e.g., amazon [3] 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, 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 [16].

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 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

[1] We use the category information as provided by the application developers. For example, Diaspora is explicitly labeled 'social-network' [8].

Table 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.

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 [5] with populated databases. We run a Chrome-based crawler [6] 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 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 [24]; 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 and 6.

3.3 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 4, we manually check every bug report. For applications with 1000 to 5000 bug reports, we randomly sample 100 bug reports 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 4.

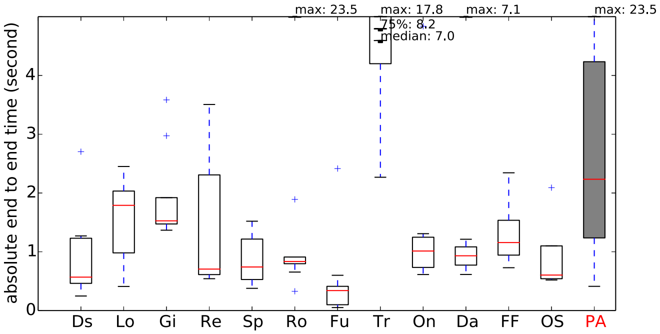


Figure 2: End-to-end page loading time

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 3.2).

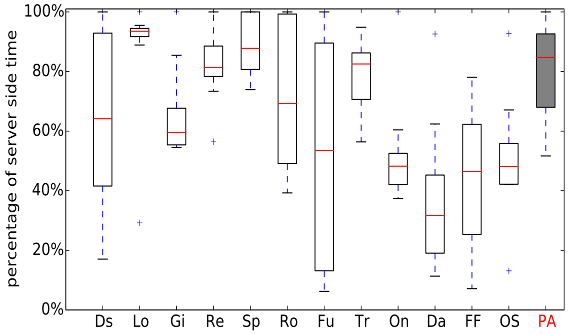


Figure 3: Percentage of server time among end-to-end time

Measured for top 10 most time-consuming pages per application. Red line: median; PA: problematic actions from all 12 applications (see Section 3.2)

3.4 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.

4 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 page loading time in Figure 2. 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.

[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 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

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 3, 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 3. This result further motivates us to study the performance problems on the server side of ORM applications.

Problematic server actions. Table 5 shows the number of problematic actions for each application identified using the methodology discussed in Section 3.2. 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. In addition, we find 64 performance issues in these 40 problematic actions, and we will discuss them in detail in Section 5.

5 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 8.

Table 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		

5.1 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.

5.1.1 *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.

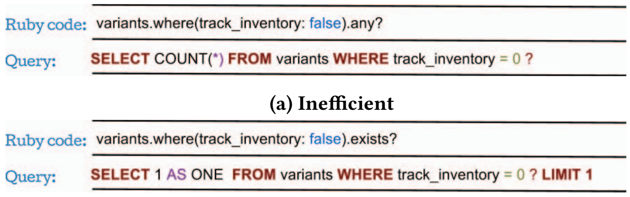
Inefficient queries. The same operation on persistent data can be implemented via different ORM calls. However, the performance of the generated queries can be drastically different. This problem has not been well studied before for ORM applications.

Figure 4 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 4(a) scans all records in the `variants` table to compute the count if no index exists, but that in Figure 4(b) 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 8 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 `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

[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.

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



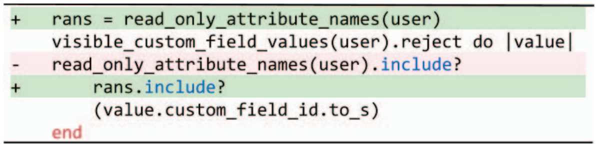
```
Ruby code: variants.where(track_inventory: false).any?
Query:     SELECT COUNT(*) FROM variants WHERE track_inventory = 0 ?
```

(a) Inefficient

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

(b) Efficient

Figure 4: Different APIs cause huge performance difference



```
+   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 5: A loop-invariant query in Redmine

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 8.

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 [29].

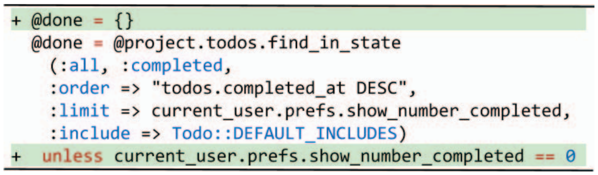
Moving computation to the server. Interestingly, there are cases where the computation should be moved to the server from the DBMS. As far as we know, this issue has not been studied before.

For example, in the patch of Spree-6819, developers replace Objects.count with Objects.size 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 8.

5.1.2 *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.



```
+ @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
```

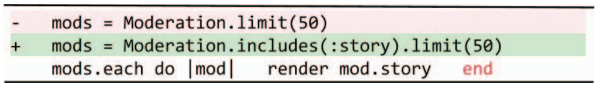
Figure 6: A query with known results in Tracks

Loop-invariant queries. Sometimes, queries are repeatedly issued to load the *same* database contents and hence are unnecessary. For instance, Figure 5 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 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.

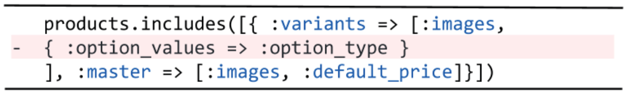
Queries with known results. A number of issues are due to issuing queries whose results are already known, hence incurring unnecessary network round trips and query processing time. An example is in Tracks-63. As shown in Figure 6, the code originally issues a query to retrieve up to show_number_completed 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 6 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.



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

Figure 7: Inefficient lazy loading in Lobsters



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

Figure 8: Inefficient eager loading in Spree

5.1.3 *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 2, 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 [7, 18, 28], we find it still prevalent: it appears in 15 problematic actions and 9 performance issues in our study.

Figure 7 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.

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 8. 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 8, 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 8) finds this to be common in the latest versions of 6 out of the 12 studied applications.

5.1.4 *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 [27] and Rails [46]. 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.

```
   milestones/index.html.haml
1: milestones.each do |milestone|
2:    render milestone
3: end
   milestones/_milestone.html.haml
4: link_to milestone.title, milestone.description
```

(a) Inefficient partial rendering

```
   milestones/index.html.haml
1: a = link_to `m_title`, `m_descr`
2: milestones.each do |milestone|
3:     a.gsub(`m_title`, milestone.title)
4:      .gsub(`m_descr`, milestone.description)
5: end
```

(b) Efficient partial rendering

Figure 9: Inefficient partial rendering in Gitlab

5.1.5 *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 9(a), to render one object and encapsulate it in a partial view file such as _milestone.html.haml in Figure 9(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 9(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 2.5× on average.

5.2 Database Design Problems

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

5.2.1 *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 [19], 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.

5.2.2 *Missing Database Indexes (MI)*

Having the appropriate indexes on tables is important for query processing and is a well-studied problem [42]. As shown in Table 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.

5.3 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.

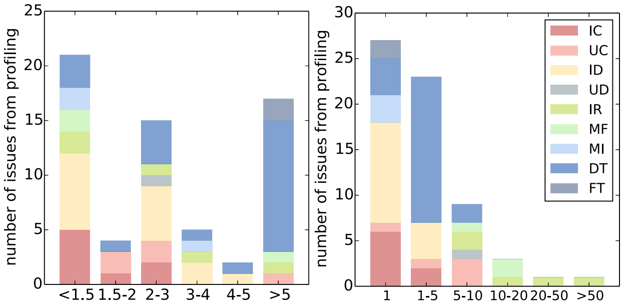
5.3.1 *Content Display Trade-offs (DT)*

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 [20] 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.

5.3.2 *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.



(a) Server-time speedup (×) **(b) Line of code changes**

Figure 10: Performance fixes and LOC involved

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 [25, 31] will help developers estimate application performance, as we will discuss in Section 8.

6 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.

6.1 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, we use the standard `will_paginate` library [20]. 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 5.3.2.

6.2 Results

In total, 64 fixes are applied across 39 problematic actions [5] to solve the 64 problems listed in Table 6.

Speedup of the fixes. Figure 10(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.

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 5.3.2.

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 5.1 and 5.2. 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 10(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 8.

7 FINDING MORE API MISUSES

Some problems described in Section 5.1 are about simple API misuses. We identify 9 such simple misuse patterns, as listed in Table 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 5.1.

As shown in Table 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.

Table 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

Only 3 out of these 428 API misuses coincide with the 64 performance problems listed in Table 6 and fixed in Section 6. 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 5.1 that cannot be detected simply through regular expression matching and will require future research to tackle.

8 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 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 [9] does, there will be fewer cases of inefficient computation, such as those discussed in Section 5.1.1. As another example, better rendering API supports could help eliminate inefficient partial render

[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.

problem discussed in Section 5.1.5. 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 5.3.

Developers will also benefit from tools that can aid in database design, such as suggesting fields to make persistent, as discussed in Section 5.2. 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 [29], query batching [28, 40], and avoiding unnecessary data retrieval [27]. There are still many automatic optimization opportunities that remain unstudied. This ranges from checking for API misuses, as we discussed in Section 7, to more sophisticated database-aware optimization techniques to remove unnecessary computation (Section 5.1.2) and inefficient queries (Section 5.1.1).

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 5.3.2. 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 5.2) and application design trade-offs (Section 5.3) naturally apply across ORMs. Most of the API use problems (Section 5.1), 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 7 may appear to be Rails specific, there are similar misuses in applications built upon Django ORM [9] 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 [21]	2	3	6	0	0	0	2	2
Zulip [23]	2	5	2	1	0	2	1	2
Django-CMS [10]	0	9	3	0	1	0	1	1

9 RELATED WORK

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 [33, 47]. Prior work [41] 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 [46] 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 [46] .

Inefficiencies in ORM applications. Previous work has addressed specific performance problems in ORM applications, such as locating unneeded column data retrieval [27], N+1 query [26], pushing computation to the DBMS [29], and query batching [28, 40, 45]. 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 [30, 33, 36, 37, 39, 43, 44]. Detecting and fixing ORM performance anti-patterns require a completely different set of techniques that understand ORM and underlying database queries.

10 CONCLUSION

Database-backed web applications are widely used and often built using ORM frameworks. We conduct a comprehensive study to understand how well such applications perform and scale with the data they manage. By profiling the latest versions of 12 representative ORM applications and studying their bug-tracking systems, we find 9 types of ORM performance anti-patterns and many performance problems in the latest versions of these applications. Our findings open up new research opportunities to develop techniques that can help developers solve performance issues in ORM applications.

11 ACKNOWLEDGEMENTS

This work is supported in part by the National Science Foundation through grants IIS-1546083, IIS-1651489, IIS-1546543, OAC-1739419, CNS-1514256, CCF-1514189, and CNS-1563788; DARPA award FA8750-16-2-0032; DOE award DE-SC0016260; gifts from Adobe, Google, and the CERES Center for Unstoppable Computing.

REFERENCES

[1] *Active Support Instrumentation.* http://guides.rubyonrails.org/active_support_instrumentation.html/.

[2] *Airbnb.* An online marketplace and hospitality service application. https://www.airbnb.com/.

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

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

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

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

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

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

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

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

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

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

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

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

[15] *Hulu.* A subscription video on demand service application. https://www.hulu.com/.

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

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

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

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

[20] *Pagination.* A library used in webpage displaying. https://github.com/mislav/will_paginate/.

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

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

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

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

[25] Michael Armbrust, Eric Liang, Tim Kraska, Armando Fox, Michael J. Franklin, and David A. Patterson. 2013. Generalized Scale Independence Through Incremental Precomputation. In *SIGMOD*. 625–636.

[26] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting Performance Anti-patterns for Applications Developed Using Object-relational Mapping. In *ICSE*. 1001–1012.

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

[28] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. 2014. Sloth: Being Lazy is a Virtue (when Issuing Database Queries). In *SIGMOD*. 931–942.

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

[30] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2008. A Scalable Technique for Characterizing the Usage of Temporaries in Framework-intensive Java Applications. In *FSE*. 59–70.

[31] Wenfei Fan, Floris Geerts, and Leonid Libkin. 2014. On Scale Independence for Querying Big Data. In *PODS*. 51–62.

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

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

[34] Emre Kiciman and Benjamin Livshits. 2007. AjaxScope: a platform for remotely monitoring the client-side behavior of Web 2.0 applications. *ACM SIGOPS Operating Systems Review.* 41, 6 (2007), 17–30.

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

[36] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes. In *ICSE*. 902–912.

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

[38] Stephen O'Grady. *The RedMonk Programming Language Rankings: June 2017.* http://redmonk.com/sogrady/2017/06/08/language-rankings-6-17/.

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

[40] Karthik Ramachandra, Chavan Mahendra, Guravannavar Ravindra, and S Sudarshan. 2015. Program Transformations for Asynchronous and Batched Query Submission. In *TKDE*. 531–544.

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

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

[43] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the Flow: Profiling Copies to Find Runtime Bloat. In *PLDI*. 419–430.

[44] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding Low-utility Data Structures. In *PLDI*. 174–186.

[45] Cong Yan and Alvin Cheung. 2016. Leveraging Lock Contention to Improve OLTP Application Performance. In *VLDB*. 444–455.

[46] Cong Yan, Junwen Yang, Alvin Cheung, and Shan Lu. 2017. Understanding Database Performance Inefficiencies in Real-world Web Applications. In *CIKM*.

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