PowerStation: Automatically detecting and fixing inefficiencies of database-backed web applications in IDE*

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

Modern web applications have stringent latency requirements while processing an ever-increasing amount of user data. To address these challenges and improve programmer productivity, Object Relational Mapping (ORM) frameworks have been developed to allow developers writing database processing code in an object-oriented manner. Despite such frameworks, prior work found that developers still struggle in developing performant ORM-based web applications. This paper presents PowerStation, a RubyMine IDE plugin for optimizing web applications developed using the Ruby on Rails ORM. Using automated static analysis, PowerStation detects ORM-related inefficiency problems and suggests fixes to developers. Our evaluation on 12 real-world applications shows that PowerStation can automatically detects 1221 performance issues across all of them. We have uploaded a tutorial on using PowerStation plugin to `https://youtu.be/v_uY5bjGuK0`.

1 INTRODUCTION

Modern web applications face stringent latency requirements and increasingly large amounts of user data to process. Recent studies have found that users expect every web page to load within two seconds [18], with one second's delay causing 11% fewer page views, a 16% decrease in customer satisfaction, and 7% loss in conversions [16]. At the same time, popular web applications often encounter user accounts increasing from a few thousands to tens of millions in few years [1, 6]. Such latency and data-scaling pressures are particularly aggravated [19, 20, 29, 30] by the pervasive use of Object-Relational Mapping (ORM) frameworks, which allow database persistent data to be manipulated through object-oriented code (e.g., Ruby on Rails [14], Django [4], and Hibernate [7]).

Previous studies have shown that developers often struggle at writing efficient web applications using ORM frameworks [19, 20, 29, 30]. Several ORM-related performance anti-patterns have been found to widely exist in real world database-backed web applications and lead to application inefficiencies. Unfortunately, many of these inefficiencies go undetected by compilers and database management systems as they focus solely on either the application code or the embedded queries, while recognizing such inefficiencies require both systems to work in tandem.

This paper presents PowerStation, an IDE plugin for Ruby on Rails (Rails) applications that automatically detects ORM-related performance problems and suggests fixes for them. It makes two contributions. First, we build a database-aware static analysis framework for Rails applications. The current framework prototype enables PowerStation to automatically detect 6 common ORM performance anti-patterns and generate patches for 5 of them. These 6 patterns have been summarized in previous work [19, 29, 30]; only 3 patterns have been automatically detected previously, and we are unaware of any tools that can automatically fix such anti-patterns.

*`http://hyperloop.cs.uchicago.edu/powerstation`

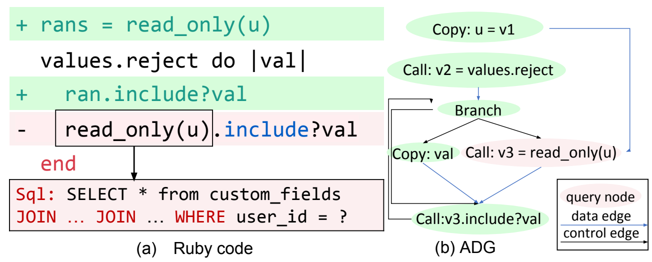

Figure 1: Loop invariant query from Redmine (the code checks which `val` in `values` list belongs to user `u`'s read-only fields)

Second, we have integrated PowerStation into a popular Rails IDE, RubyMine [15], so that Rails developers can easily benefit from PowerStation to improve the efficiency of their applications. The source code of PowerStation is available on GitHub [10].

2 PERFORMANCE ANTI-PATTERNS

PowerStation currently tackles six performance anti-patterns. While these patterns have been extracted by previous work [19, 29, 30] 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) [30]. A query is repeatedly issued in every iteration of a loop to load the *same* database content. In the real-world example shown in Figure 1a, hoisting the query out of the loop can speed up the application by more than 10× [13].

Dead store queries (DS) [30]. SQL queries are repeatedly issued to assign the same memory object with different database content, without any use of the memory object in between, making all but the last query unnecessary.

Unused data-retrieval queries (RD) [19, 30]. 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) [29]. Queries with common sub-expressions are issued, causing unnecessary re-computation.

API misuses (IA) [30]. Different ORM APIs can be used to retrieve the same results from the database, but they can differ drastically in terms of performance. For example, the two Rails code snippets in Figure 2 both check if a `user` owns any blog posts. However, they use different APIs, `count` versus `exist`, that are translated

```
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 2: API misuse from Onebody (the upper code is less efficient than the lower code)

to different SQL queries by Rails: `select count` versus `select limit 1` — the former scans all the records in table `blogs` with specified `user_id`, counts the number of records, and checks (in the Ruby application) whether the count is greater than 0. Meanwhile, the latter returns immediately as soon as it finds one record with the specific `user_id`. The latter can easily improve the resulting application performance by 1.7× [30].

Inefficient data rendering (IR) [30]. 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 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 by their applications 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 be used to detect such patterns.

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.

3.1 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 first inlines function calls to enable inter-procedural analysis. This process involves type inference [22], as Ruby is dynamically typed, and identifying Ruby code that is implicitly invoked by an controller action through view rendering, Rails callback functions, `ActiveRecord` validation functions, etc.

Program dependency graph (PDG) generation. PowerStation uses JRuby to process above pre-processed source code, and then builds the PDG from JRuby intermediate representation (IR). As illustrated in Figure 1b, every node n in the PDG represents a statement in the JRuby IR. Every edge e in PDG represents either control

[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.

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: Analyzing table schemas

dependency or data dependency. A data-dependency edge $n_1 \to n_2$ 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 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, 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, PowerStation learns about the schemas of table `users` and `blogs` 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) explicitly invoked raw SQL queries through Base.connection.execute. Any query identified above is represented as a Query node in the ADG.[2]

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 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

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

```
+ 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)

```
<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>
...
```

(b)

Figure 4: Fix inefficient rendering (gsub is a string substitution API, replacing its 1st parameter with 2nd parameter)

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 [29]. 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 [30] 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, assigning 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 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 particularly important for Dead Store and Unused Data cases, where data uses from a different action, which is rare, may make PowerStation fixes invalid—developers can check before accepting PowerStation fixes. In all other cases, PowerStation's suggested fixes do preserve program semantics.

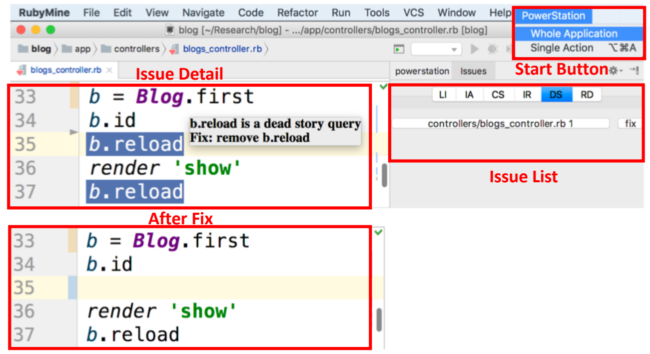

Figure 5: Screenshots of PowerStation IDE Plugin

4 POWERSTATION IDE INTEGRATION

4.1 PowerStation IDE plugin features

We have implemented PowerStation as an IDE plugin for RubyMine [15], a popular IDE for Ruby on Rails. A screenshot of PowerStation is shown in Figure 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 PowerStation plugin tutorial [9].

Issues list. The right panel, as highlighted in Figure 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 in a project are listed. Users can also choose to display inefficiencies of a particularly type as shown in Figure 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 a file button in the issue list will navigate users to the corresponding file in the editor, with the inefficient code highlighted. When a user hovers her mouse over the highlighted code, the inefficiency reason will be displayed, as shown in Figure 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, and display the fixed code in the editor panel. At that point, the original "fix" button becomes an "undo" button, allowing users to revert the fix if needed.

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 5). This string is instantiated with program variables and expressions output from PowerStation static analysis, and displayed using IntelliJ API `FileDocumentManager`.

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

Table 1: Inefficiencies detected by PowerStation in 12 apps

App.	Loop Invariants	Unused 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

5 EVALUATION

PowerStation can be downloaded from IntelliJ plugin repository [5] and easily installed in RubyMine.

We have evaluated PowerStation using the latest versions of 12 open-source Ruby on Rails applications, including the top two popular applications on Github of 6 categories, Forum, Collaboration, E-commerce, Task-management, Social Network, and Map. As shown in Table 1, PowerStation automatically identifies 1221 inefficiency issues, and automatically generates 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 12–625 seconds to analyze the entire application that ranges from 4k to 145k lines of code in our experiments. Developers can also choose to analyze one action at a time, which usually takes less than 10 seconds in our experiments.

6 RELATED WORK

Recent work used static program analysis to find optimization opportunities in database-related applications, such as QBS [20] for query synthesis, QURO [28] for query reordering in transactions, and PipeGen [23] for automatic data pipe generation. None of these techniques detect or fix anti-patterns addressed by PowerStation.

Much work was done to detect performance issues, like loop inefficiency, data bloating, under-utilized data structure, and cache false-sharing, in compiler research [21, 24–27]. As pointed out in previous work [19, 29, 30], database-backed web applications suffer from different types of inefficiencies and require new detection and fixing tools that are aware of queries issued by the application.

Dynamic profiling tools have been built for Rails applications [11]; they cannot statically detect and fix inefficiency root causes. Static analysis tools have been built to detect code smells [12], security vulnerabilities [2], and code cleaning opportunities [3, 17] in Rails applications. However, they do not detect performance problems. The Intellij platform [8] itself provides more than 1000 plugins for RubyMine, while none of them tackles code inefficiency.

7 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.

REFERENCES

[1] *Airbnb Data Growth*. https://www.recode.net/2017/7/19/15949782/.
[2] *BRAKEMAN*. https://github.com/presidentbeef/brakeman.
[3] *DEADWEIGHT*. https://github.com/aanand/deadweight/.
[4] *Django*. https://www.djangoproject.com/.
[5] *Download PowerStation Plugin*. https://plugins.jetbrains.com/plugin/10604-powerstation/.
[6] *Facebook*. https://www.statista.com/statistics/346167/facebook-global-dau/.
[7] *Hibernate*. http://hibernate.org/.
[8] *IntelliJ*. http://www.jetbrains.org/intellij/sdk/docs/welcome.html/.
[9] *PowerStation*. www.hyperloop.cs.uchicago.edu/powerstation/.
[10] *PowerStation Static Analysis Framework/*. https://github.com/hyperloop-rails/powerstation/.
[11] *rackminiprofiler*. https://github.com/MiniProfiler/rack-mini-profiler/.
[12] *RAILS-BEST-PRACTICES*. https://github.com/flyerhzm/rails_best_practices/.
[13] *Redmine-23334*. https://redmine.org/issues/23334/.
[14] *Ruby on Rails*. http://rubyonrails.org/.
[15] *RubyMine*. https://www.jetbrains.com/ruby/.
[16] *Speed is a killter*. https://blog.kissmetrics.com/speed-is-a-killer/.
[17] *TRACEROUTE*. https://github.com/amatsuda/traceroute/.
[18] *What is page load time and why is it important?* https://www.bigcommerce.com/ecommerce-answers/what-page-load-time-and-why-it-important/.
[19] 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. *Transactions on Software Engineering* (2016).
[20] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *PLDI*.
[21] 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*.
[22] Michael Furr, Jong-hoon David An, Jeffrey S Foster, and Michael Hicks. 2009. Static type inference for Ruby. In *SAC*.
[23] Brandon Haynes, Alvin Cheung, and Magdalena Balazinska. 2016. Pipegen: Data pipe generator for hybrid analytics. In *SOCC*.
[24] Tongping Liu and Emery D Berger. 2011. Sheriff: precise detection and automatic mitigation of false sharing. In *OOPSLA*.
[25] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting performance problems via similar memory-access patterns. In *ICSE*.
[26] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static detection of asymptotic performance bugs in collection traversals. In *PLDI*.
[27] Guoqing (Harry) Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding low-utility data structures. In *PLDI*.
[28] Cong Yan and Alvin Cheung. 2016. Leveraging lock contention to improve OLTP application performance. *VLDB* (2016).
[29] Cong Yan, Junwen Yang, Alvin Cheung, and Shan Lu. 2017. Understanding Database Performance Inefficiencies in Real-world Web Applications. In *CIKM*.
[30] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *ICSE*.