An Industrial Experience Report on Performance-Aware Refactoring on a Database-centric Web Application

Boyuan Chen and Zhen Ming (Jack) Jiang
York University
Toronto, Canada
{chenfsd, zmjiang}@eecs.yorku.ca

Paul Matos and Michael Lacaria
Copywell Inc.
Toronto, Canada
{paulm, michael}@2ics.com

***Abstract*—Modern web applications rely heavily on databases to query and update information. To ease the development efforts, Object Relational Mapping (ORM) frameworks provide an abstraction for developers to manage databases by writing in the same Object-Oriented programming languages. Prior studies have shown that there are various types of performance issues caused by inefficient accesses to databases via different ORM frameworks (e.g., Hibernate and ActiveRecord). However, it is not clear whether the reported performance anti-patterns (common performance issues) can be generalizable across various frameworks. In particular, there is no study focusing on detecting performance issues for applications written in PHP, which is the choice of programming languages for the majority (79%) of web applications. In this experience paper, we detail our process on conducting performance-aware refactoring of an industrial web application written in Laravel, the most popular web framework in PHP. We have derived a complete catalog of 17 performance anti-patterns based on prior research and our experimentation. We have found that some of the reported anti-patterns and refactoring techniques are framework or programming language specific, whereas others are general. The performance impact of the anti-pattern instances are highly dependent on the actual usage context (workload and database settings). When communicating the performance differences before and after refactoring, the results of the complex statistical analysis may be sometimes confusing. Instead, developers usually prefer more intuitive measures like percentage improvement. Experiments show that our refactoring techniques can reduce the response time up to 93.0% and 93.4% for the industrial and the open source application under various scenarios.**

I. Introduction

Many of the modern web applications are database-centric systems, which extensively use the databases in order to accomplish users' tasks. There are many existing database-centric web applications, ranging from enterprise (e.g., content management, and enterprise resource planning) to consumer applications (e.g., e-commerce, and discussion forums). To ease the development processes, Object-Relational Mapping (ORM) has been introduced as a middle layer between the application code and the database. It enables developers to manage the database using the same Object-Oriented programming languages and automatically translate the application source code into underlying SQL queries. On one hand, the ORM layer provides a nice conceptual abstraction and improves code readability, so that developers can focus on their application logic instead of the underlying database accesses. On the other hand, the translation process is not transparent, and can often lead to sub-optimal performance in both open source [1], [2] and commercial applications [3], [4].

Software performance is one of the crucial factors related to the success of web applications, as failing to do so would result in customers' abandonment and loss of revenue [5]. There have been many prior studies that focus on detecting and deriving performance anti-patterns (a.k.a., common performance issues) in the ORM layer. However, they are generally focused on a particular framework (e.g., Hibernate for Java Spring framework [1], [4] or ActiveRecord for Ruby on Rails [2], [6], [7]). It is not clear whether the reported performance anti-patterns can be generalizable across different frameworks or programming languages, as there are many ORM frameworks available. In particular, there are no studies focusing on applications written in PHP, which is the choice of programming languages for 79% of web applications [8]. Instead of writing PHP code from scratch, developers usually prefer building their applications based on existing web frameworks, which have a set of features (e.g., ORM, task scheduling, and message queues) implemented already. Among all the PHP web frameworks, Laravel is currently the most popular one [9]. The ORM framework inside Laravel is called *Eloquent*. It is important to study the performance issues across different ORM frameworks due to the following three reasons:

- **Different default behavior**: Depending on the frameworks, the default ORM behavior can be very different. For example, to find the first object which satisfies a predicate, the default behavior for ActiveRecord (`where().first()`) is to sort the results by their primary keys followed by fetching the first object, whereas the default Eloquent behavior (`where()->first()`) is to only retrieve the first object without sorting. Hence, to ensure the quality of various database-centric web applications, we need to not only detect general performance anti-patterns, but also framework-specific ones [3].
- **Different ORM configurations**: Some ORM configurations are related to the representation (e.g., mapping

specific classes to database tables), and others are related to their runtime behavior (e.g., caching or data loading). Different frameworks have different ways to specify these configurations. For example, Hibernate uses configuration files and annotations, whereas Laravel uses API calls. The differences in the configurations would require different detection and optimization techniques.

- **Different analysis tools**: All the existing ORM performance anti-pattern detection works (e.g., [1], [2], [6]) leveraged program analysis. Such techniques are generally framework and programming language dependent. To support a different programming language, a new parser would be needed for the static analysis. Similarly, a different instrumentation framework is needed for the dynamic profiling tools.

In this paper, we report our experience on conducting performance-aware refactoring on a database-centric industrial application written in Laravel. It is performance-aware refactoring, as our approach detects and refactors performance issues without altering the functional behavior of the applications under study. The contributions of this paper are:

1) This is the first research work which focuses on performance-aware refactoring in Laravel and PHP.
2) We built a catalog of 17 performance anti-patterns by studying the prior research works on other programming languages or frameworks and by experimentation on our industrial application. Among them, two new anti-patterns were discovered by us.
3) Certain anti-patterns were framework/programming language specific, whereas others are general. The performance impact of the individual anti-pattern instance may vary significantly based on the actual usage context(workload and database settings).
4) We have applied our performance-aware refactoring approach to the industrial and one popular open source application. The results show that we can reduce the response time up to 93.4% for these two applications. The developed techniques have already been adopted and used daily by our industrial partners.
5) Our process and experience will be useful for researchers and practitioners who are interested in optimizing the performance of database centric applications.

Paper Organization. The rest of this paper is structured as follows. Section II explains the Laravel framework and provides some background information about our industrial application. Section III explains our approach using a running example. Section IV explains the refactoring techniques for detected anti-patterns. Section V evaluates the performance improvement. Section VI discusses lessons learnt. Section VII surveys related work. Section VIII describes the threats to validity. Finally, Section IX concludes the paper.

II. Background

In this section, we provide a background about Laravel (Section II-A) and our studied industrial application (Section II-B).

A. Laravel

Laravel is a very popular web framework for PHP. It contains various libraries like an ORM framework (Eloquent) and a packaging system. The applications developed under Laravel follows the Model-View-Controller (MVC) pattern. Figure 1 shows an example Laravel application. In this example, the user makes a request to the `/orders` page. Once the web server receives this request, it looks through the routing file `web.php` to locate the corresponding method (the method `show`) in the controller (`OrderController`). The controller (`OrderController`) then interacts with the model by invoking the `Order::activeCount()` method. This method accesses the database through Eloquent APIs (e.g., `where` and `count`), which will translate the PHP code into the corresponding database query and pass it to the database. The query is executed and the Eloquent ORM passes the result back to the model class and subsequently to the controller. The result is stored in the `$active_orders` variable in the controller. This variable is then rendered to the view file (`index.blade.php`) as the `$active` variable. The computed response is sent back and displayed in the user's browser. In addition to the ORM APIs from Eloquent, Laravel also provides developers with the choice of querying the database using raw SQL queries in the code.

B. Our Industrial Application Under Study

SP is a complex web-based Consumer Relational Management (CRM) written in Laravel. It is used for stream-lining the process of quoting and ordering printing jobs for consumers. Depending on the jobs, many different configuration options (e.g., choice of ink, colour of the paper, types of printers) are needed and specified. Pricing of orders also vary depending on the configured values (e.g., paper and ink costs as well as due dates). Once jobs are created, proofs are generated and sent to customers for approval. Historical jobs and their configurations are also stored, so that recurrent customers or orders can be conveniently made. Currently the database of SP is around 1 GB and consists of more than 90 tables and over a million records. One of the main issues with SP is that some pages take a long time to load. This problem is getting even worse over time, since the size of the database is constantly growing ($\sim 10\%$ every month). In the next two sections, we will discuss about our approach to detecting and refactoring the reported performance issues.

III. Detecting Performance Issues

In this section, we explain our approach to detecting performance issues in SP. It is also applicable to other database-centric web applications written in Laravel. As illustrated in Figure 2, our approach consists of five steps: (1) literature study, (2) dynamic analysis, (3) static analysis, (4) filtering, and (5) deriving new performance anti-patterns.

In the remaining part of this section, we will explain these steps using a running example, which has two page requests: `/activeorders` and `/outsources`. Both requests are routed to the same controller (`OrderController.php`),

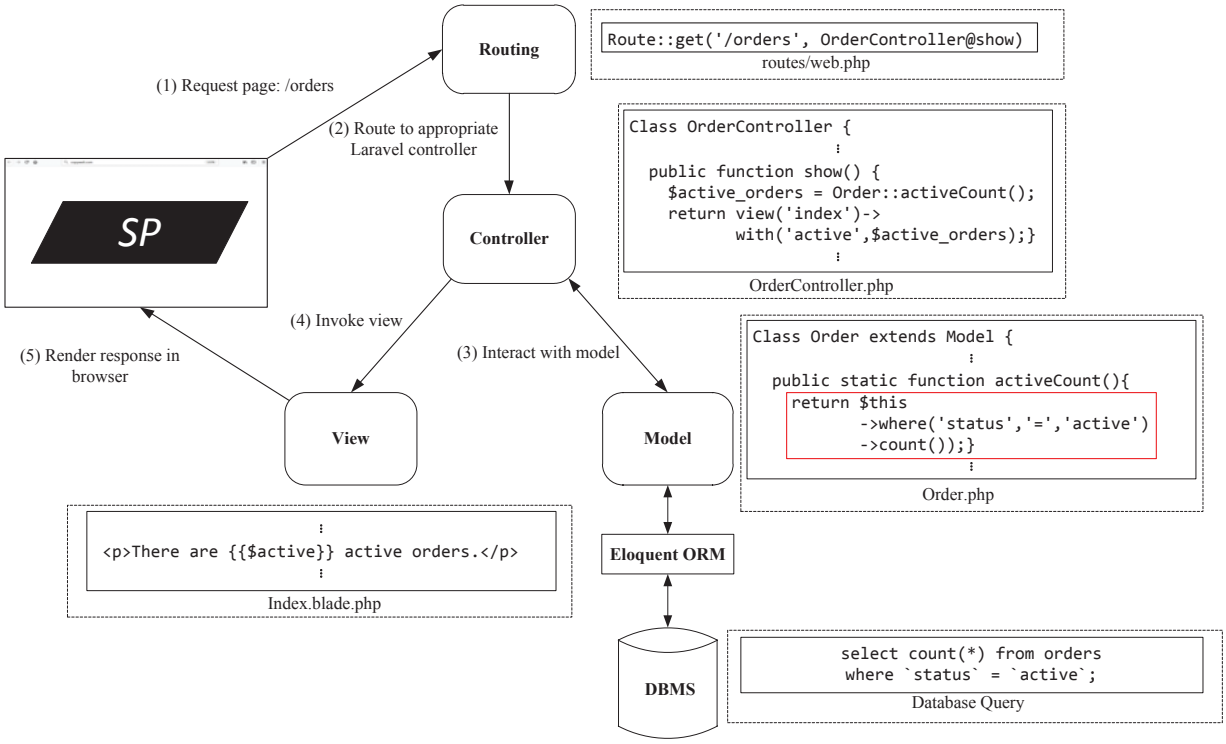

Fig. 1: An example of a Laravel-based web application written in PHP.

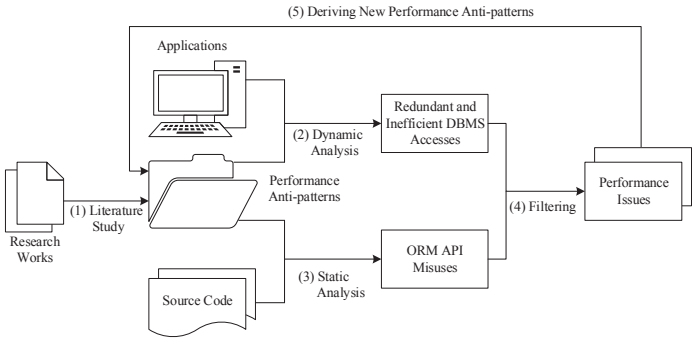

Fig. 2: Our process of detecting performance issues in Laravel-based database-centric web applications.

but are handled by two different functions: `showActive` and `showOutsource`. The `showActive` function in the controller subsequently calls the `active` function from the model (`Order.php`). To save space, we only showed the code snippets for the controller (`OrderController.php`) and the model (`Order.php`) in Figure 3(a). For the first page request, the `active` function in the `Order.php` retrieves all the `orders` with `active` status and sorts them by the order ID (`id`) at line 40. Then it retrieves three particular attributes from the sorted orders: `price`, `cost`, and `progress` at line 41, 42, and 43. The above data retrieval processes invoke several Eloquent APIs (`where`, `orderBy`, and `pluck`) and the third column of Figure 3(b) shows the resulting SQL queries. In Eloquent, line 40 only returns a `Builder` object. Developers need to invoke API calls like `get` (executing the query) or `pluck` (retrieving a column) to execute this query. Hence, calling `pluck` at lines 41, 42, and 43 result in three similar SQL queries, except selecting different fields in the table. For the second page request, the `showOutsource` function from `OrderController.php` checks whether there are any outsourced orders at line 81. Since the `where` API call is followed by the `get` call, executing this line result in a `select` query as shown in the later part of Figure 3(b).

Step 1 - Literature study

We went through the existing works [1], [2], [3], [6], [10], which focus on detecting performance anti-patterns in the database-centric applications. We did not include [11], as it focuses on view-centric performance optimization, which alters the functional behavior, whereas our focus is on performance-aware refactoring. As a result, we compiled a list of performance anti-patterns, their generalizability, and the suggested refactoring techniques.

Step 2 - Dynamic Analysis

Some of the reported anti-patterns need to be detected by dynamic analysis. There have been different approaches to collecting the runtime traces from the applications. Most of them are framework (e.g., the Rails Active Support Instrumentation APIs [2]) or programming language specific (e.g., AspectJ [10]). In addition, the detection methods vary from dynamic taint analysis [10] to threshold checking [2]. However, all the existing dynamically detected anti-patterns

OrderController.php

```
40 public function showActive() {
⋮                                ⋮
51   $activeNumber = Order::active($orders);
⋮                                ⋮
70 public function showOutsource() {
⋮                                ⋮
81   if (count(OutSource::where('type','order')->get()) > 0)
82     $this->countOutsource();
```

Order.php

```
10 class Order extends Model {
⋮                                ⋮
30   public static function active() {
⋮                                ⋮
40     $allStats = Order::where('status','active')->orderBy('id','asc');
41     $price = $allStats->pluck('price');
42     $cost = $allStats->pluck('cost');
43     $progress = $allStats->pluck('progress');
```

(a) Source code.

Page requested	Code path	SQL	Duration
/activeorders	OrderController.php:showActive() -> OrderController.php:51 -> Order.php:41	`select` `price` `from orders where` `status` = `active` `order by id ASC`	101.3ms
	OrderController.php:showActive() -> OrderController.php:51 -> Order.php:42	`select` `cost` `from orders where` `status` = `active` `order by id ASC`	102.7ms
	OrderController.php:showActive() -> OrderController.php:51 -> Order.php:43	`select` `progress` `from orders where` `status` = `active` `order by id ASC`	100.0ms
/outsources	OrderController.php:showOutSource() -> OrderController.php:81	`select * from outsources where` `type` = `order`	2.1 ms
	⋮	⋮	⋮

(b) Processed data from Clockwork.

Fig. 3: Our running example.

share one common theme: focusing on locating inefficient or redundant database computations.

We encoded the user usage using *JMeter* [12], which is an open source performance testing tool, so that the same tests can be replicated and compared across different versions of the application. To avoid measurement bias and errors [13], each scenario is repeated at least 30 times. JMeter records the end-to-end response time for each scenario. However, the timing for each scenario can be broken down into various components like the network transmission time, the time spent on loading and displaying the results (i.e., page rendering time), the web server processing time, and the database query processing time. To further isolate the timing between these components, we used a open source profiler called Clockwork [14].

Clockwork is a profiler for Laravel-based PHP applications. Clockwork collects various dynamic information (e.g., the request URLs, the executed source code, the invoked database queries and their durations) for each page request. Clockwork can be installed as a browser plugin, so that the profiling data can be viewed within a browser. It also stores the profiled information in the web server as a JSON file for each individual request. In this paper, we leverage the recorded JSON files, as they can be automatically processed. We used another open source tool called *Page Load Time* [15] to track the end-to-end response time, and the page rendering time.

For each scenario, we correlated the timing results from *Page Loading Time* and the profiling data from *Clockwork*. Based on our analysis, we found that most of the time consuming steps for SP were about processing the application logic (a.k.a., within the controller, the model, and the database side). The time spent on the network transmission and the page rendering was generally very small. Some of the function calls may be time-consuming. But they were actually performing the necessary computation for the complex user requests and should not considered as performance issues. We would like to only identify problematic areas which are either *redundant* (i.e., the same PHP methods or database queries are executed multiple times back-to-back) or *inefficient* (i.e., very similar database queries are executed multiple times).

We processed the Clockwork data to recover the call paths for each database query. We grouped them based on the requested pages. Figure 3(b) shows the processed Clockwork data for our running example. The first column refers to the pages requested. Under the same page request, different code paths are executed. They are shown in the second column of that table. The resulting SQL queries and their execution time are shown in the third and fourth column. For the `/activeorders` page request, there are three queries executed due to the `pluck` calls at line 41, 42, and 43. This is inefficient, as the only differences among these queries are the selected columns. A more efficient solution is to add `->get()` at the end of line 40, as it will execute a query to select all columns and store them in the variable `allStats`. In this way, the following three `pluck` calls will only retrieve the values of the attributes from the memory. No further database queries are invoked.

Step 3 - Static Analysis

The remaining performance anti-patterns derived from the prior research are detected statically. These anti-patterns are mostly ORM API misuses and complement nicely with the results of our dynamic analysis. As our dynamic analysis is focusing on redundant or inefficient database queries spanning over multiple lines or even source code files, static analysis can pin-point other code-specific issues. In our running example from Figure 3(a), line 81 in `OrderController.php` contains an anti-pattern related to the ORM API misuses. The `if` statement is to check whether there are any outsourced orders. Calling the `get()` API will cause the database to return all the records in the `outsource` table which has the `type` value to be `order`. This query can be slow if there are many records which match this criteria. A more efficient approach would be to replace the `get()` method with the `first()` method, which only returns the first matching record.

Since such static anti-patterns were originally studied and specified in other programming languages (e.g., Ruby [2] and Java [1]), we searched for the corresponding problematic Eloquent APIs from the Laravel documentation. In order to automatically detect these issues, we implemented a static code checker, which scans through the PHP code. Our static checker first parses the source code files into Abstract Syntax Trees (ASTs) by using a popular open source tool: *PHP*

parser [16]. Then we detected these static anti-patterns by traversing through ASTs and matching them with various regular expressions. In our running example, our checker will extract the condition checks for every `if` statement by analyzing the ASTs. Among all the condition checks, this code snippet is flagged in our tool, as it matches with a regular expression (`count.*get`).

Step 4 - Filtering

We found that some of the reported code snippets, especially the ones detected using our static analysis, are dependent on the actual usage context and may not cause performance problems for the end users. For example, some of the detected code snippets might not be executed, or the amount of computing (e.g., # of database records involved) is very small in the actual customer usage context. Since developers have limited time, we would like to only pin-point the problematic areas which really impact the user experience. Hence, in this step, we only included the issues existed in the problematic scenarios. Similar to [2], we considered a scenario as problematic, if the end-to-end response time exceeds two seconds.

In our running example, the page `/activeorders` takes 2.5 second and the page `/outsources` takes 0.5 seconds. Hence, the issue flagged at line 81 of `OrderController.php` was filtered out. We only reported the issue at line 40 of `Order.php` to the developers.

Step 5 - Deriving New Performance Anti-patterns

We reported a set of user impacting performance issues to the developers in step 4. For issues which are not detected by the existing static anti-patterns, we added them into our catalog of performance anti-patterns. The reported performance issue in our running example did not belong to any of the existing performance anti-patterns. We classified them as a new anti-pattern: **Mid-result misuse** and added it to our catalog. For the dynamically detected anti-patterns, we also encoded them into our static analysis tool so that we can automatically categorize all the reported issues and apply the same suggested refactoring techniques on them.

As shown in the third column of Table I, we ended up with 17 performance anti-patterns in total. For each anti-pattern, we assigned them a category, an ID, and a name. We intentionally used the name of the Eloquent APIs (e.g., `Eloquent:get` in `AP-10`) in some of the anti-pattern names if they are related to the API misuses. We used the Eloquent APIs, if the anti-patterns are general or specific to Eloquent, as that's the focus of this paper. For other anti-patterns, we labeled the applicable ORM frameworks to avoid confusion.

Among the 17 anti-patterns, two (`AP-04` and `AP-05`) are newly discovered in this paper. There are 11 general anti-patterns, which are applicable across different ORM frameworks. There are five anti-patterns only applicable for ActiveRecord (in Ruby); and one anti-pattern only applicable for Eloquent (in PHP). These ORM specific anti-patterns correspond to some particular API calls in the framework. Among all 17 performance anti-patterns, there are 12 applicable to Laravel. For the ones that are not applicable to Laravel, it is mostly due to differences in the APIs across frameworks. For example, `AP-13` is related to the default behavior of `first` in ActiveRecord. This API automatically sorts the results using the primary key and only returns the first row as the result. Such API does not exist in Eloquent. We also showed the refactoring techniques for each anti-pattern as well as their origin (a.k.a., the prior research works which reported them). Using the approach described above, we were able to detect all of the performance anti-patterns applicable to PHP. The next section describes the refactoring techniques.

```
foreach($tasks as $task) {
    $count = $order::all()->count();
    // other statements
… }
```

(a) Inefficient

```
$count= $order::all()->count();
foreach($tasks as $task) {
    // other statements
… }
```

(b) Efficient

Fig. 4: An example of [AP-01] Loop-invariant queries.

IV. Performance-Aware Refactoring

Although there are various techniques to improve the performance of these detected performance issues, we only focused on the performance-aware refactoring techniques, as we intended to minimize the risks of functional behavior changes for the end-users. We used the following process to synthesize the refactoring suggestions: for anti-patterns which were originally studied in the prior research, we tried to apply the same refactoring techniques to the problematic PHP code. If the refactoring technique involves API changes, we tried to look for similar APIs in Laravel. If the refactorings technique involves algorithm changes, we tried to re-implement them in PHP. For newly discovered anti-patterns, we consulted the developers of SP on how to refactor them. We verified the validity of all the refactoring techniques by comparing the performance before and after the refactorings. Different frameworks may require different refactoring techniques. For example, in Hibernate, `AP-06` can be refactored by adding annotations or changing settings in configuration files [1]. In Laravel, such refactoring technique does not apply due to differences in the ORM configurations. In the rest of this section, we will describe the refactoring technique for the performance anti-pattern shown in Table I. We will skip `AP-13` to `AP-17`, as they are not applicable to PHP.

[AP-01] `Loop-invariant queries`

Description: This anti-pattern is about repeatedly executing the same queries, which return the same results. These queries are redundant. Figure 4(a) shows such an example. The loop iterates through all tasks. During each iteration, it always queries the total number of orders, whose results never change.

TABLE I: The complete catalog of performance anti-patterns.

Category	ID	Anti-pattern	General/ ORM specific	Refactoring	Origin
Unnecessary Computation	AP-01	Loop-invariant queries	General	Introducing intermediate variable	[2]
	AP-02	Dead-store queries	General	Deleting unnecessary statement	[1], [2], [10]
	AP-03	Queries with known results	General	Involving default behavior	[2]
	AP-04	Redundant condition check	General	Introducing intermediate variable	New
	AP-05	Mid-result misuse	Eloquent	Incurring queries in advance	New
Inefficient Data Accessing	AP-06	Inefficient lazy loading	General	Changing loading configuration	[1], [2], [3], [10]
	AP-07	Inefficient eager loading	General	Changing loading configuration	[2]
	AP-08	Inefficient updating	General	Applying batch updating	[2]
Unnecessary Data Retrieval	AP-09	Eloquent: map	General	Eloquent: pluck	[2], [3], [6], [10]
Inefficient Computation	AP-10	Eloquent: get	General	Eloquent: first	[2]
	AP-11	Eloquent: array_sum.*pluck	General	Eloquent: pluck.*sum	[2], [6]
	AP-12	Eloquent: orderBy.*count	General	Deleting unnecessary ordering	[2]
	AP-13	ActiveRecord: where.first	ActiveRecord	ActiveRecord: find_by	[2]
	AP-14	ActiveRecord: pluck+pluck	ActiveRecord	ActiveRecord: union	[2]
	AP-15	ActiveRecord: if exists?	ActiveRecord	ActiveRecord: find_or_create_by	[2]
	AP-16	ActiveRecord: .count	ActiveRecord	ActiveRecord: .size	[2]
Inefficient Rendering	AP-17	ActiveRecord: link_to	ActiveRecord	ActiveRecord: gsub	[2]

```
$count = $order::all()->count();
$count = $order::where("status","=","active")->count();
```

(a) Inefficient

```
$count = $order::where("status","=","active")->count();
```

(b) Efficient

Fig. 5: An example of [AP-02] Dead-store queries.

```
$oldTask= Task::where("type","=", "inactive")
          ->take($setting_number)->get();
```

(a) Inefficient

```
if ($setting_number != 0) {
  $oldTask= Task::where("type","=", "inactive")
          ->take($setting_number)->get();
  ...}
```

(b) Efficient

Fig. 6: An example of [AP-03] Queries with known results.

Refactoring: The refactoring technique for this anti-pattern is to move the query out of the loop and assign the queried result to an intermediate variable as shown in Figure 4(b). The performance after the refactoring should improve due to the removal of the redundant queries.

[AP-02] Dead-store queries

Description: This anti-pattern is about storing the results of multiple queries into the same variable, some of which are never accessed. This causes redundancy in the computation. Figure 5(a) shows such an example. The count value computed from the first query is never used.

Refactoring: The refactoring technique for this anti-pattern is to remove the query, whose results are used in the subsequently computation. The performance after the refactoring should improve as the redundant queries no longer exist.

[AP-03] Queries with known results

Description: This anti-pattern checks if there are redundant queries, whose results are known to be empty before computation. Figure 6(a) shows such an example. It invokes `take` to return only a pre-specified number of records, by the variable `$setting_number` from the query. This query will be redundant, if `$setting_number` is set to be zero.

Refactoring: The refactoring technique for this anti-pattern is to add a condition check as shown in Figure 6(b). The database query will be executed, if the value of `$setting_number` is larger than zero. The performance after the refactoring should improve as it prevents unnecessary queries.

[AP-04] Redundant condition check *NEW*

Description: This anti-pattern is about introducing redundant queries in condition checks. It is a new anti-pattern that we have found. Figure 7(a) shows one such example. The `if` statement checks whether an order has any related customers. If the condition is true, the variable `$cus` will be assigned with the name of the first customer. Under the true branch of the condition check, the same queries are executed twice, which is unnecessary.

Refactoring: The refactoring technique for this anti-pattern is to assign the query result to an intermediate variable `$tmp`

```
if ($order->customer()->first()) {
    $cus = $order->customer()->first()->name; ...}
```

(a) Inefficient

```
$tmp = $order->customer()->first();
if ($tmp) {
    $cus = $tmp->name; ... }
```

(b) Efficient

Fig. 7: An example of [AP-04] Redundant condition check.

```
$books = Book::all();
foreach ($books as $book) {
    $name = $book->author->name;}
```

(a) Lazy-loading

```
$books = Book::with("authors")->all();
foreach ($books as $book) {
    $name = $book->author->name;}
```

(b) Eager-loading

Fig. 8: An example of [AP-06/07] Inefficient lazy/eager loading.

```
foreach(Order::all() as $order) {
    $order->status = "complete";
    $order->save();}
```

(a) Inefficient

```
Order::all()->update(["status" => "complete"]);
```

(b) Efficient

Fig. 9: An example of [AP-08] Inefficient updating.

```
$users = User::all()->map(function($user)
                        {return $user->name;})
select * from users;
```

(a) Inefficient

```
$users = User::all()->pluck("name");
select name from users;
```

(b) Efficient

Fig. 10: An example of [AP-09] `map`.

shown in Figure 7(b). The performance after the refactoring should improve due to the removal of the duplicated query.

[AP-05] `Mid-result misuse` `*NEW*`

Description: This anti-pattern is about applying redundant queries on the same `Builder` object. It is a new anti-pattern that we have found. Line 40 in Figure 3(a) shows one such example. In Eloquent, method `where` does not incur the actual query. It only generates an instance of `Builder` object. Developers need to invoke APIs like `get` (executing the query) or `pluck` (retrieving a column) to invoke and execute the query. In this example, the three `pluck` invocations are not needed, as they can leverage the results from the same query.

Refactoring: The refactoring technique for this anti-pattern is to add the `get` API call after `where` to invoke and execute the query. The performance after the refactoring should improve as only one query is needed to retrieve the results.

[AP-06] `Inefficient lazy loading`

Description: This anti-pattern is about not eager loading the relational data when needed. Figure 8(a) shows one such example. It first retrieves all the records from `Book`. Then it iterates through each book to retrieve the names of the authors. It will result in N+1 queries: 1 query for extracting all the books, N queries for extracting the names of the authors for each book. Such computations can be done more efficiently by eager loading all the related data.

Refactoring: There are three ways to refactor this anti-pattern: (1) to eager load the related data in the first query by adding `with` call inline as shown in Figure 8(b); or (2) to eager load the related data by adding a class attribute `$with=['authors'];` or (3) to conduct a left join on the `books` table with the `authors` table. The performance after the refactoring should improve as only two queries are executed instead of N+1.

[AP-07] `Inefficient eager loading`

Description: This anti-pattern is the opposite of `AP-06`. If the associated objects are too large, eager loading everything will create memory bloat, which can lead to the slow down of the web applications due to memory paging. This may also cause the application to be unresponsive.

Refactoring: The refactoring technique for this anti-pattern is to delay the eager loading till later. As shown in Figure 8(a), we should not use the `with` call if the associated author objects are too large. The performance of the application should improve as the data intensive computation is delayed.

[AP-08] `Inefficient updating`

Description: This anti-pattern is about updating records in a loop. Figure 9(a) shows one such example. It iterates through all the `orders` to update the `status` to be `complete`. This would cause performance issues as it would incur `N` queries, where `N` is the number of records.

Refacotring: The refactoring technique for this anti-pattern is to update records in a batch as shown in Figure 9(b). The performance after refactoring should improve as the number of queries is reduced to one instead of `N`.

[AP-09] `Eloquent: map`

Description: This anti-pattern is about using `map` method in PHP to apply a function on a collection of objects. If this function involves with database queries, this might result in unnecessary computations. Figure 10(a) shows one such example. It first retrieves all the records from the `User` table. Then, it gets their names by invoking the `map` method.

Refactoring: The refactoring technique is to use `pluck` to get specific columns as shown in Figure 10(b). The performance

```
array_sum($bill->pluck("total")->toArray());
select `total` from bill;
```

(a) Inefficient

```
$bill->pluck("total")->sum()
select sum(`total`) from bill;
```

(b) Efficient

Fig. 11: An example of [AP-11] `array_sum.*pluck`.

after refactoring should improve as the database query will only return the specified column(s) instead of the entire rows.

[AP-10] `Eloquent: get`

Description: This anti-pattern is about checking whether one or multiple rows exist in the database with certain predicates. Line 81 of `OrderController.php` in Figure 3(a) shows such an example. It checks whether there are outsourced orders. This can cause performance issues, as the `get()` method will have the database return all the records in the `outsources` table with type `order`.

Refactoring: The refactoring technique for this anti-pattern is to replace the `get()` API call with `first()`, which only returns the first record from the result set. If there are 500 records in the `outsource` table with `order` as their `type`. Calling `first()` will only return one record, compared to returning all 500 records by calling `get()`.

[AP-11] `Eloquent: array_sum.*pluck`

Description: This anti-pattern is about using `pluck` to retrieve the attributes from the queried results and computing the sum on the web server. Large amount of data will be sent from the database to the web server for computing the sum. This will cause unnecessary overhead, as the database can perform the addition efficiently on their end. Figure 11(a) shows such an example. The inefficient code first retrieves the `total` column of all the records from the `bill` table. The results are transmitted to the web server, which subsequently computes the sum.

Refactoring: The refactoring technique for this anti-pattern is to apply `sum` directly after `pluck` in the code snippet as shown in Figure 11(b). The performance after the refactoring should improve as the web server does not need to receive the large volume of the queried results and perform the computation afterwards.

[AP-12] `Eloquent: orderBy.*count`

Description: This anti-pattern is about counting the total number of sorted entries in the queried results. The sorting step is unnecessary, as it has nothing to do with the counting. Figure 12(a) shows an example. The `count` method is called after the `orderBy` method.

Refactoring: The refactoring technique for this anti-pattern is to remove the `orderBy` method call in the code snippet as shown in Figure 12(b). The performance after the refactoring should improve due to the elimination of the sorting process.

```
Order::where("status","=","active")->orderBy("id")->count();
select count(*) from orders
where `status`=`active` order by `id`;
```

(a) Inefficient

```
Order::where("status","=","active")->count();
select count(*) from orders where `status`=`active`;
```

(b) Efficient

Fig. 12: An example of [AP-12] `orderBy.*count`.

TABLE II: The evaluation results for SP. Due to page limitation, we only showed the anti-patterns, if they existed in SP. We marked one anti-pattern in bold, if $p - value < 0.05$ and the effect size is medium or above.

ID	# of instances	effect size	% reduced time
AP-04	**> 100**	**medium - large**	**2.9% - 21.3%**
AP-05	**2**	**large**	**24.2% - 69.0%**
AP-06	**> 10**	**large**	**17.0% - 93.0%**
AP-08	**> 10**	**large**	**70.1% - 80.2%**
AP-10	> 10	N/A	N/A
AP-11	1	N/A	N/A

V. Evaluation

In this section, we evaluate the effectiveness of our performance-aware refactoring approach.

A. Approach

For each anti-pattern instance (a.k.a, reported performance issues), we compared the performance of SP before and after applying the suggested refactoring technique. JMeter [12] was used to automate the performance benchmarking process. It executes the pre-defined user workload and records the end-to-end response time for each scenario. We used JMeter to repeatedly executed each scenario for at least 30 times to avoid measurement bias and errors [13].

To assess the impact of the refactoring technique, each time we only applied refactoring on one anti-pattern instance. We reset the database and rebooted the web server to restore the test environment before each JMeter test. We performed the non-parametric Wilcoxon rank-sum test (WRS) to compare the response time before and after refactoring. In addition, to quantify the magnitude of the performance improvement, we calculated the effect sizes using a non-parametric technique called the Cliff's Delta (CD) [17].

B. Result

Although WRS and CD techniques provided a statistical rigorous view of the performance differences, the developers of SP prefer a more intuitive representation. In particular, they would like to know how much faster the performance improves after the refactoring. Hence, we just calculate the percentage of improvement before and after refactoring.

Table II shows the performance assessment results for all the performance anti-patterns that we detected and refactored in SP. Fixing instances from AP-10 did not have much performance impact. It is meaningless to fix the one instance from AP-11, as it was dead code. In total, four out of the six detected anti-patterns were considered as impactful in SP.

AP-04 is the most common anti-pattern in SP with over one hundred occurrences. More than ten occurrences of AP-06 and AP-08 were also reported. There were two occurrences of AP-05. Due to confidentiality concerns, we did not show the exact number of anti-pattern instances here. The differences in response time before and after refactoring are statistically significant for all four anti-patterns with medium to large effect sizes. However, the performance improvement across different instances of the anti-patterns vary significantly. For example, the biggest improvement (93.0%) comes from refactoring one instance of AP-06. The response time was 5.96 seconds and 0.42 seconds before and after the refactoring. The impact of refactoring on some other instances of AP-06 is not as big. For example, refactoring another instance of AP-06 only yields 17.0% improvement (4.59 seconds vs. 3.81 seconds). The smallest performance improvement is from AP-04, reducing 2.9% - 21.3% response time after refactoring.

We have reported the above anti-pattern instances and their refactoring suggestions to the developers of SP. They happily accepted our suggestions and applied the changes in their code base. After applying these changes, the response time of SP has been reduced from 2.9% to 93.0% across different scenarios. One of the comments we received from the developers of SP said: *"you have definitely hit the nail on the head with what you have found so far."*. This clearly demonstrated the impact of our approach.

Summary: The statically and dynamically detected anti-patterns can have large performance impact on SP. Although some anti-patterns are pervasive, they may not impact the performance much based on the actual usage context. The performance improvement due to refactoring may vary significantly across instances of the same anti-patterns.

VI. Discussion and Lessons Learned

In this section, we discuss the lessons learned during our process on detecting and refactoring performance anti-patterns.

***Lesson 1**: Techniques on detecting and refactoring anti-patterns in database-centric web applications.*

We derived the catalog of performance anti-patterns based on prior works in other programming languages and our experience while working on SP. Based on our experience, the exsiting technique on detecting and refactoring anti-patterns [1], [2] in database-centric web applications can be applied to a language and ORM framework which have not been applied to. To demonstrate the generalizablility of this catalog, we also applied our approach on another Laravel-based web application: *Cachet* [18]. It is the open source project with the most stars in GitHub written in Laravel. Cachet is a status page application, which is used to communicate downtime and system outages to users. Since there are no databases which provide the realistic usage context, we varied the size of different tables while keeping the same number of records in the database to be 20,000. We chose 20,000 in this study, as a previous work [2] found that it is a realistic setting for many database-centric open source applications. We found five types of anti-pattern instances (AP-02, AP-05, AP-06, AP-10, and AP-12), which are reported in at least one of the database settings. We measured the performance impact after refactoring the anti-pattern instances. As a result, we achieved up to 93.4% improvement in Cachet.

***Lesson 2**: The importance of the usage context for detecting and refactoring anti-pattern instances.*

For the Cachet experiment mentioned above, we found if we distributed the contents among the tables differently, the performance will vary significantly even if the overall size of the database kept constant. Here we describe the experiments we have conducted on the homepage of Cachet. Using our approach, there are four types of anti-patterns detected. However, by varying the number of records in these two tables: components and component_groups, some anti-patterns did not impact the experience of the end users. We kept the sum of the records in these two tables as 4,000. In one experiment we set the number of records in these two tables as 2,000 each, while in another experiment we changed them to 1,000 and 3,000. We tried many combinations and measured the resulting performance. Figure 13 shows the distribution of the response time for these experiments. Under some experiments, the homepage is fast (1.2 seconds), whereas for other experiments the response time can be as high as 3.7 seconds.

The above experiments and our experience on SP clearly demonstrate the importance of the usage context (workload and database setting) while detecting and refactoring anti-pattern instances. The developers of SP provided us with a sanitized version of the production database and the user workload. For applications (e.g., open source or newly developed applications) which do not have such information, it is important to include the experimented workload and database settings when reporting detected anti-pattern instances.

***Lesson 3**: Filtering and presenting anti-pattern instances to the developers.*

Since there can be many anti-pattern instances and developers have limited time, we only kept the detected anti-pattern instances if the scenarios under test impact the end-user experiences. Same as in [2], we chose two seconds as our threshold. Since we believe if the performance of certain scenarios were poor, refactoring any or all the reported anti-pattern instances should be helpful. However, based on our experiments in SP and Cachet, refactoring some anti-patterns may have little or even negative performance impactFor example, after refactoring some instances of AP-10 in Cachet, we received negative performance impact (2.41 vs. 2.42 seconds). Therefore, one of

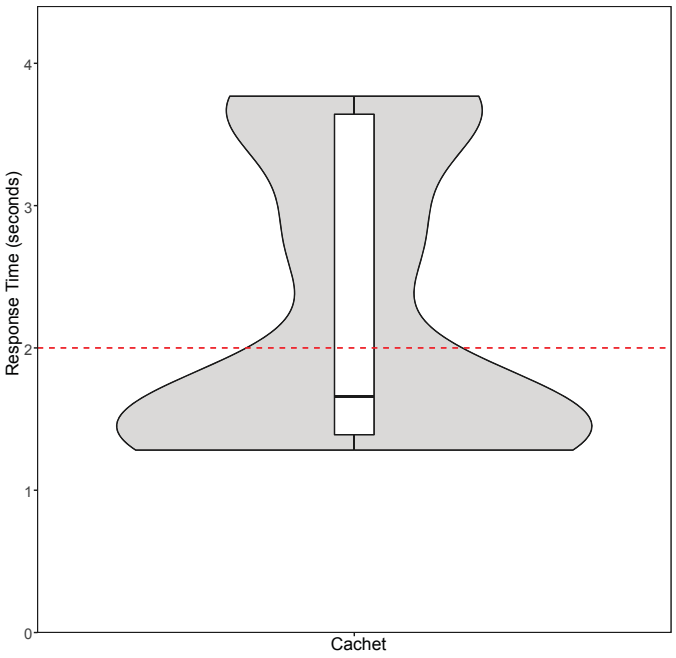

Fig. 13: Response time distribution for the homepage of Cachet by varying the number of records in different tables while keeping the total number of records the same.

the common questions we got from the developers of SP was on the anticipated performance impact of these anti-pattern instances. There is currently one technique, which executes and times the anticipated SQL queries after the refactoring [1], [10]. We are currently incorporating this technique in our approach. We will first get the estimated timing for various anti-pattern instances after refactoring. Different scenarios may invoke code snippets, which correspond to the same anti-pattern instances. We intend to compute a subset of suggested anti-pattern instances using search optimization techniques. This subset would only contain the smallest number of anti-pattern instances, which have the biggest anticipated performance improvement.

***Lesson 4**: Assessing the impact of the refactoring technique.*

When evaluating the performance impact of various refactoring techniques, the developers of SP found that there is a gap between the perceived differences by human and the statistical differences. WRS test and CD have been used quite often [1], [10], [19] in the past research works to compare and quantify the differences between two distributions of data. However, in our study, we have found that in some cases even an improvement of $30msec$ can lead to a medium or large effect sizes. For example, refactoring one instance of `AP-10` can reduce the average response time from 2.00 seconds to 1.97 seconds. Although the effect size for such change is large, the differences can hardly be noticed by users. Therefore, we also calculated the percentage of improvement as a way to capture performance changes perceived by humans.

***Lesson 5**: Static versus dynamic analysis.*

We detected performance anti-patterns using both the static and the dynamic techniques. Static analysis is easy to apply to the whole code base, but not all of the reported issues from them are relevant. In SP, the static analysis reported over 100 instances of `AP-04`. However, only a handful really impacted the user experience. Dynamic analysis is more expensive, as we need to set up test environments and analyze the profiled information. However, it can be used to discover new anti-patterns. The two new anti-patterns `AP-04` and `AP-05` reported in this paper were discovered via dynamic analysis. After we discovered the new anti-pattern, we encoded them into our static detection. In addition to detecting anti-pattern instances, another objective of the static analysis in our approach is to automatically categorize the dynamically detected performance issues. The corresponding refactoring techniques can be suggested subsequently, instead of manually analyzing the individual code snippets and thinking of ways to improve them.

VII. Related Work

In this section, we discuss two areas of the prior research related to our work: (1) performance issues in database-centric applications; and (2) testing database-centric applications.

A. Performance Issues in Database-centric Applications.

Smith et al. [20], [21] were the first to propose a catalog for various performance anti-patterns. Some of these anti-patterns were related to the ways how the applications performed the database queries. The database-centric applications were implemented differently by then, as raw SQLs queries were usually specified in the application source code. To characterize and detect ORM anti-patterns, Chen et al [1], [3], [10] studied the performance anti-patterns for applications implemented under the Hibernate framework in Java. Yang et al. [2], [6] analyzed the bug reports and executed performance tests for 12 open source Ruby on Rails applications. They have synthesized nine types of ORM-level anti-patterns. They also developed an IDE-plugin to automatically detect and fix these anti-patterns [7]. Different from the prior works, our work analyzed the applications written in Laravel. Our catalog of ORM-level anti-patterns was built by studying these prior works and applying them to Laravel-based open source and industrial applications. During this process, we also found two additional performance anti-patterns. We also reported a series of lessons learned based on the challenges that we have faced and the feedback gathered from the developers.

B. Testing Database-centric Applications

Mcminn et al. [22] and Kapfhammer et al. [23] proposed a set of test coverage and adequacy criteria to assess the quality of test suites for database-centric applications. Grechanik et al. [24], [25] proposed new approaches to preventing database deadlocks and automatically reproducing deadlocks for testing purposes. Our work is related to Taneja et al. [26], [27], in which they discussed the obstacles and challenges

of anonymizing the production databases for the outsourced testing teams. They showed that the application's functional behavior (code coverage) can vary if the database fields were not properly sanitized. Our work is focused on the performance aspects related to the different settings of the database. While keeping the same number of records in the database, we have shown that the application performance can vary significantly by changing the number of records among different tables.

VIII. Threats to Validity

A. External Validity

We have only demonstrated the generalization of our approach to Laravel-based applications by applying it on one industry and one open source application. There maybe additional performance anti-patterns unreported and our refactoring techniques can be just one of the many solutions to improve the performance of these anti-patterns.

B. Construct Validity

During the anti-pattern detection phase, we leveraged the Clockwork data to: (1) find the mappings between the source code and their corresponding database queries, and (2) to gather the response time on invididual database queries. Such information is accurate, as the monitoring overhead for Clockwork, although high ($> 10\%$), is mainly imposed on the web server and has little impact on the database processing and the page rendering. We turned off Clockwork and only collected the end-to-end response time from the JMeter when assessing the performance impact of various refactoring techniques.

C. Internal Validity

One scenario may contain multiple anti-pattern instances. The performance of the same anti-pattern instances may vary significantly based on the usage context. When assessing the performance impact of different refactoring techniques, each time we only fixed one anti-pattern instance and measured its response time. We restored the database and restarted the web server for each experiment. This process was repeated for all the anti-pattern instances.

IX. Conclusions

Databases play a key role in modern web applications. Many developers leverage ORMs to access databases to avoid directly interacting with them. However, such setup may not yield optimal performance. In this paper, we reported our experience on conducting performance-aware refactoring on database-centric web applications written in Laravel. Our complete catalog of performance anti-patterns, which consists of 15 existing and 2 new anti-patterns, were derived from existing research literatures and dynamic analysis on SP. Experiments show that refactoring these anti-pattern instances may result in up to 93.4% response time reduced in SP and Cachet.

References

[1] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1001–1012.

[2] J. Yang, P. Subramaniam, S. Lu, C. Yan, and A. Cheung, "How Not to Structure Your Database-backed Web Applications: A Study of Performance Bugs in the Wild," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, 2018.

[3] T.-H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting problems in the database access code of large scale systems: An industrial experience report," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 71–80.

[4] D. Maplesden, K. von Randow, E. Tempero, J. Hosking, and J. Grundy, "Performance Analysis Using Subsuming Methods: An Industrial Case Study," in *IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE SEIP)*, 2015.

[5] K. Eaton, "How One Second Could Cost Amazon $1.6 Billion In Sales," https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales, Last accessed 05/01/2019.

[6] C. Yan, A. Cheung, J. Yang, and S. Lu, "Understanding Database Performance Inefficiencies in Real-world Web Applications," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, ser. CIKM '17. New York, NY, USA: ACM, 2017, pp. 1299–1308.

[7] J. Yang, C. Yan, P. Subramaniam, S. Lu, and A. Cheung, "Powerstation: automatically detecting and fixing inefficiencies of database-backed web applications in IDE," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 884–887.

[8] "Usage statistics and market share of PHP for websites," https://w3techs.com/technologies/details/pl-php/all/all, Last accessed 04/16/2019.

[9] "Top 10 PHP Frameworks," https://stackify.com/php-frameworks-development/, Last accessed: 05/02/2019.

[10] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks," *IEEE Trans. Softw. Eng.*, vol. 42, no. 12, pp. 1148–1161, Dec. 2016. [Online]. Available: https://doi.org/10.1109/TSE.2016.2553039

[11] J. Yang, C. Yan, C. cheng Wan, S. Lu, and A. Cheung, "View-centric performance optimization for database-backed web applications," in *To appear in the Proceedings of the 41th International Conference on Software Engineering*, ser. ICSE '19, 2019.

[12] "Apache Jmeter," http://jmeter.apache.org/, Last accessed 04/10/2019.

[13] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically Rigorous Java Performance Evaluation," in *Proceedings of the 22nd International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2007.

[14] "Clockwork - PHP dev tools integrated to your browser," https://underground.works/clockwork, Last accessed 03/20/2019.

[15] "Page load time," https://chrome.google.com/webstore/detail/page-load-time/fploionmjgeclbkemipmkogoaohcdbig, Last accessed 05/12/2019.

[16] "PHP-Parser - A PHP parser written in PHP," https://github.com/nikic/PHP-Parser, Last accessed 04/18/2019.

[17] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys?" in *Annual meeting of the Florida Association of Institutional Research*, 2006.

[18] "Cachet: an open source status page system for everyone," https://github.com/CachetHQ/Cachet, Last accessed 05/09/2019.

[19] R. Gao, Z. M. Jiang, C. Barna, and M. Litoiu, "A framework to evaluate the effectiveness of different load testing analysis techniques," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, April 2016, pp. 22–32.

[20] C. U. Smith and L. G. Williams, "New software performance antipatterns: More ways to shoot yourself in the foot," in *Computer Measurement Group (CMG) Conference*, 2002.

[21] ——, "More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot," in *Computer Measurement Group (CMG) Conference*, 2003.

[22] P. Mcminn, C. J. Wright, and G. M. Kapfhammer, "The effectiveness of test coverage criteria for relational database schema integrity constraints," *ACM Transactions on Software Engineering Methodology (TOSEM)*, 2015.

[23] G. M. Kapfhammer and M. L. Soffa, "A family of test adequacy criteria for database-driven applications," in *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-11. New York, NY, USA: ACM, 2003, pp. 98–107. [Online]. Available: http://doi.acm.org/10.1145/940071.940086

[24] M. Grechanik, B. M. M. Hossain, and U. Buy, "Testing Database-Centric Applications for Causes of Database Deadlocks," in *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)*, 2013.

[25] M. Grechanik, B. M. M. Hossain, U. Buy, and H. Wang, "Preventing database deadlocks in applications," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013.

[26] K. Taneja, M. Grechanik, R. Ghani, and T. Xie, "Testing Software in Age of Data Privacy: A Balancing Act," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*, 2011.

[27] B. L. . M. G. . D. Poshyvanyk, "Sanitizing and Minimizing Databases for Software Application Test Outsourcing," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST)*, 2014.