Understanding Database Performance Inefficiencies in Real-world Web Applications

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

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

ABSTRACT

Many modern database-backed web applications are built upon Object Relational Mapping (ORM) frameworks. While such frameworks ease application development by abstracting persistent data as objects, such convenience comes with a performance cost. In this paper, we studied 27 real-world open-source applications built on top of the popular Ruby on Rails ORM framework, with the goal to understand the database-related performance inefficiencies in these applications. We discovered a number of inefficiencies ranging from physical design issues to how queries are expressed in the application code. We applied static program analysis to identify and measure how prevalent these issues are, then suggested techniques to alleviate these issues and measured the potential performance gain as a result. These techniques significantly reduce database query time (up to 91%) and the webpage response time (up to 98%). Our study provides guidance to the design of future database engines and ORM frameworks to support database application that are performant yet without sacrificing programmability.

1 INTRODUCTION

Object-relational mapping (ORM) frameworks are widely used to construct applications that interact with database management systems (DBMSs). While the implementations of such frameworks vary (e.g., Ruby on Rails [15], Django [5], and Hibernate [10]), the design principles and goals remain the same: rather than embedding SQL queries into the application code, ORMs let developers manipulate persistent data as if it is in-memory objects via APIs exposed by the ORMs [1, 6]. When executed, such API calls are translated by the ORM into queries executed by the DBMS and the query results are serialized into objects returned to the application. By raising the level of abstraction, this approach allows developers to implement their entire application in a single programming language, thereby enhancing code readability and maintainability.

However, the increase in programming productivity comes at a cost. With the details of query processing hidden, programmers often do not understand how their code is translated to queries and how queries are executed. Furthermore, lacking an understanding of application semantics makes it difficult for the ORM and DBMS to optimize how to manipulate persistent application data. Both aspects make applications built atop ORMs vulnerable to performance problems that impact overall user experience, as we have observed from the issue reports for such ORM applications [9].

To understand the causes of performance issues in ORM applications, we studied 27 real-world applications built using the popular Ruby on Rails framework. The applications are all under active development and are chosen to cover a wide variety of domains: online forums, e-commerce, collaboration platforms, etc. Our goal is to understand database-related inefficiencies in these applications and their causes. To the best of our knowledge, this is the first comprehensive study of database-related inefficiencies in database-backed applications built using ORM frameworks.

In our study, we found a number of issues across these applications that cause performance problems, ranging from how data is stored in the DBMS to how queries are expressed in the application code. Most of these issues have not been reported in prior work. Furthermore, we implemented a number of static program analysis to systematically quantify how common the issues that we identified are across different applications. We proposed a number of possible optimizations to resolve these issues that require no or little effort from developers. As case studies, we applied these optimizations manually to some of these applications and our evaluation showed that these optimizations improved performance by up to 41×.

In summary, this paper makes the following contributions:

- We performed the first comprehensive database-performance study of real-world applications built upon ORM frameworks. We chose 27 applications, covering a wide variety of domains. Our results show that many applications share similar performance inefficiencies, including poor physical database design, coding patterns that lead to inefficient queries being generated by the ORMs, and the lack of caching that results in redundant computation.
- For each performance inefficiency, we implemented an automated static analysis to systematically measure how prevalent it is across different applications.
- For each inefficiency, we proposed solutions that open up new research opportunities. We manually implemented a number of these optimizations and showed that they provided up to 41× performance improvement in our cases studies.

In the following, we review the design of ORMs and web applications built with ORMs in Section 2. Then in Section 3, we describe our study methodology. We discuss our findings and optimizations in Section 4 and Section 5, followed by related work in Section 6.

CIKM'17, , November 6–10, 2017, Singapore.

ACM ISBN 978-1-4503-4918-5/17/11…$15.00
https://doi.org/10.1145/3132847.3132954

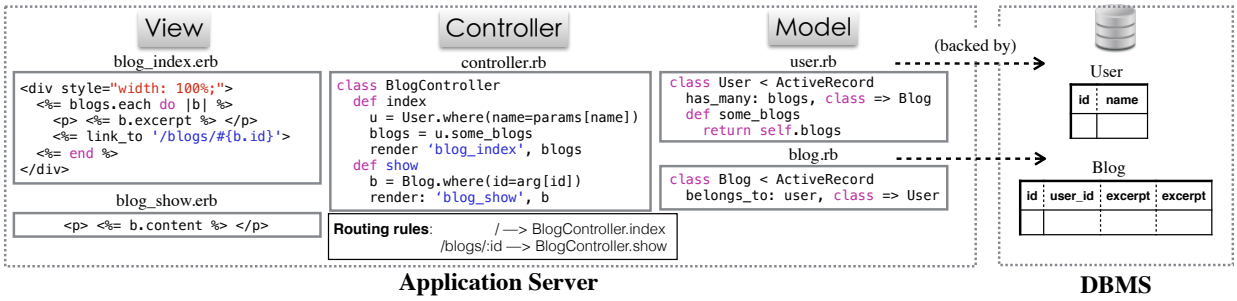

Figure 1: Rails application example abridged from publify [13]

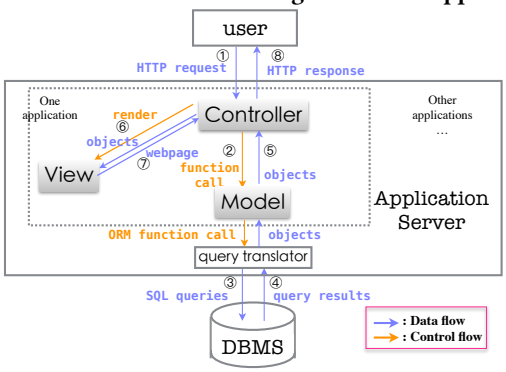

Figure 2: Architecture of a Rails application

2 BACKGROUND

In this section we provide an overview of applications constructed using ORM frameworks, using Ruby on Rails ("Rails") as an example. Rails is an ORM framework that is widely used in building web applications, including many well-known websites (e.g., Airbnb, Hulu, etc.). Applications constructed on top of Rails share similar architecture as those built using other ORMs such as Hibernate [10] and Django [5].

2.1 Design of Rails applications

Figure 2 shows the architecture of a typical Rails application. The application is hosted on a Rails application server that communicates with a DBMS to retrieve and manipulate persistent data. Internally, a Rails application consists of three components: model, view, and controller [30]. Upon receiving a user's request, say to render a web page (① in Figure 2), the Rails server invokes the corresponding *action* that resides in the appropriate *controller* based on the routing rules provided by the application, as shown in Figure 1. During its execution, the controller interacts with the DBMS by invoking ORM functions provided by Rails ②, which Rails translates into SQL queries ③. The query results ④ are serialized into *model* objects returned to the controller ⑤, and subsequently passed to the *view* ⑥ to construct a webpage ⑦ to be returned ⑧.

As an illustration, Figure 1 shows the code of a blogging application. While the controller and the model are written in Ruby, the view code is written in a mix of ruby and mark-up languages such as HTML. There are two actions defined in the controller: `index` and `show`. Inside the `index` action, the Rails functions `User.where` and `User.some_blogs` are translated to SQL queries at run time to retrieve blog records from the DBMS. The retrieved records are then passed to `render` to construct a webpage defined by the view file `blog_index.erb`. This webpage displays the excerpt and the URL link (`link_to`) of every blog. Clicking the link brings the user to a separate page, with another action (`show`) called with the corresponding blog identifier to retrieve the corresponding blog details via another query issued by `Blog.where`.

2.2 Object relational mapping in Rails

Like other ORMs, Rails by default maps each model class hierarchy to a table. Each field in the model class maps to a column in the table. Hence, for the code shown in Figure 1, `User` objects are stored in the `User` table in the DBMS, and likewise for `Blog` objects. Developers define relationships among the model classes, such as `belongs_to` and `has_many` [7]. Such relationships are implemented using foreign keys or by building a separate table storing the relationship. For example, in Figure 1, each `Blog` object `belongs_to` one `User`, hence Rails stores the unique `user_id` associated with each `Blog`. As we will see, how classes and their relationships are modeled can affect application performance significantly.

3 ANALYSIS METHODOLOGY

We now describe how we analyze applications, our profiling methodology, and the application corpus we choose for the study.

3.1 Static program analysis

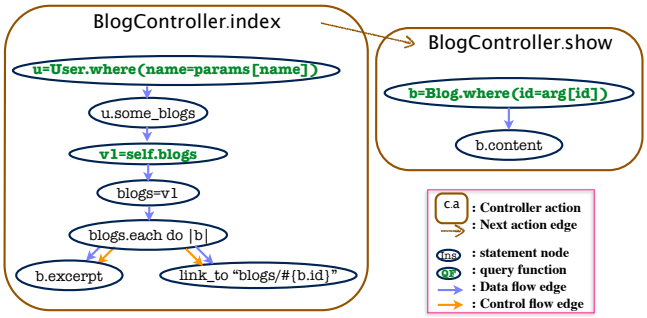

Figure 3: Action Flow Graph (AFG) example

We build an ORM-aware static analyzer for Rails applications. It produces an *Action Flow Graph* (AFG) that contains the data-flow and control-flow information for each action, along with ORM-specific information inside and across different actions. For instance, the graph labels query function nodes that we identify based on Rails semantics, such as the `where` function shown in Figure 3. To

support inter-procedural analysis, we start building AFGs from the top-level function that runs an action and iteratively inline all function calls.[1] In addition to regular function calls, we also inline filters and validations [7], which are functions that are automatically executed before an action and before any function that modifies database, respectively.

The AFG contains *next action* edges between pairs of actions (c_1, c_2) if c_2 can be invoked from c_1 as a result of a user interaction. To identify such interactions, we identify actions that render URLs (e.g., `link_to` in Figure 1) or forms in their output webpages, and determine the subsequent actions that may be triggered as a result of an user interaction (e.g., clicking an URL or submitting a form) based on the application routing rules, such as those listed in Figure 1.

3.2 Profiling applications

In addition to static analysis, we profile a number of real-world Rails applications using synthetic data to evaluate the potential of our proposed optimizations.

As we do not have the exact real-world workload for the applications, we make our best effort in collecting real-world workload statistics and build our synthetic workload accordingly. Particularly, we collect workload statistics such as the total number of posts (projects) in a forum (project-management) website, the average number of posts (projects) by each user, the distribution of these numbers, and others. The sources of our statistics include (1) websites running these applications (e.g., lobsters [12] has a running website, similarly for redmine [14]), (2) websites running an application similar to the one in our corpus (e.g., we use statistics collected from diaspora [3] for sugar [16]. Both are forum applications and share similar functionalities); (3) users' reports about their deployment experience of an application [9].

After populating the database, we visit each page generated by the application and record the response time taken to generate each response page, breaking it down into the time spent on the Rails server running ruby code and the time spent on DBMS executing queries.

3.3 Application corpus

We conduct a comprehensive study of 27 open-source Rails applications, selected from github based on their popularity (80% of them have more than 200 stars), the number of contributors (88% of them have more than 10 contributors), the number of commits, and the application category. Details about them are shown in Table 1. They cover a broad range of characteristics in terms of DBMS usage: transaction-heavy (e.g., e-commerce), read-intensive (e.g., social networking), or write-intensive (e.g., blogging, forum); and in terms of application complexity: small applications with simple functionality (e.g., kandan [11], a small chatting-room application that only supports chatting and file sharing), or large applications with many features (e.g., gitlab [8], a website to manage and share git repositories). We believe that these represent all major categories of ORM-based applications. For findings presented in following sections, we take the average of all actions for each application (shown in figures), and then average across all applications (statistics in observation boxes), unless specified separately.

[1]We assume all recursive calls terminate with call depth 1.

We profile seven representative applications with the most stars from the five categories mentioned above and use them in our case study to evaluate the potential of each optimization we propose. For each of these applications, we deploy the Rails server and MySQL or PostgresSQL database on one AWS node with 4 CPUs, each with 16GB of memory, and run client on local PC browser for profiling.

4 SINGLE ACTION OPTIMIZATIONS

In this section, we present our findings on performance issues within one single action. The causes of these issues are mainly poor translation from ORM API to database queries, and rendering too much data from query results.

4.1 Query translation

Many ORM frameworks allow users to construct queries by chaining multiple function calls (e.g., `where`, `join`), with each chain translated into a SQL query. We find that current query translation scheme often generates inefficient queries. However, they can be optimized by understanding the query generation process and how the results are subsequently used. In the following we introduce common types of inefficient queries, propose ways to identify and optimize them, and manually implement a subset of the optimizations to demonstrate potential performance gain.

4.1.1 Caching common subexpressions. By studying the query log, we find that many queries share common subexpressions that cause repetitive computation. An example is shown in Listing 1. First, `subPj` is created on Line 1 from `projects` by issuing a query (`Q1` in Figure 4) to retrieve the first level nested `projects` of the given `projects`. Then, Line 2 creates `descPj` also from `projects`, issuing another query (`Q2` in Figure 4) to get all descendants of the given `projects`. These two queries share the same selection predicate and the results are both ordered by project `id`, i.e., they share the same common subexpression.

```
subPj = projects.children.visible.order(`id ASC')
descPj = projects.where(`lft > ? AND rgt < ?')
                 .visible.order(`id ASC')
```

Listing 1: Example queries that partially share predicate, abridged from redmine [14].

	Original queries	Query time
Q1:	SELECT projects.* FROM projects WHERE projects.parent_id=? AND (projects.status <> 9) ORDER BY projects.id ASC	0.70 sec
Q2:	SELECT projects.* FROM projects WHERE (projects.lft > ? AND projects.rgt < ?) AND (projects.status <> 9) ORDER BY projects.id ASC	2.91 sec
	Simulating intermediate-query-result cache	
Q3:	CREATE VIEW pj AS SELECT * FROM projects WHERE (projects.status <> 9) ORDER BY projects.id ASC	0.04 sec
Q4:	SELECT * FROM pj WHERE pj.parent_id = ?	0.69 sec
Q5:	SELECT * FROM pj WHERE (pj.lft > ? AND pj.rgt < ?)	0.47 sec

Figure 4: Performance gain by caching of query results.

To systematically measure the number of common subexpressions, we generate the AFG of each action and compare the query call chains using static analysis: if the same query function is used in two different query function chains, the corresponding queries will share a common subexpression. Using this analysis, we count the number of queries that share subexpressions with a previous query issued within the same action. The result is shown in Figure 5.

App	Loc	App	Loc	App	Loc	App	Loc	App	Loc	App	Loc
forum		**social networking**		**collaboration**		**task management**		**resource sharing**		**e-Commerce**	
forem	5957	kandan	1694	redmine	27589	kanban	2027	boxroom	2614	piggybak	2325
lobsters	7127	onebody	32467	rucksack	8388	fulcrum	4663	brevidy	13672	shoppe	5904
linuxfr	11231	commEng	34486	railscollab	12743	tracks	23129	wallgig	11189	amahi	8412
sugar	11738	diaspora	47474	jobsworth	15419	calagator	1328	enki	5275	sharetribe	67169
				gitlab	145351			publify	16269		

Table 1: Application categories and lines of source code (all applications are hosted on github). All applications listed are studied with static analysis. Among them, gitlab, kandan, lobsters, publify, redmine, sugar, tracks are profiled as well.

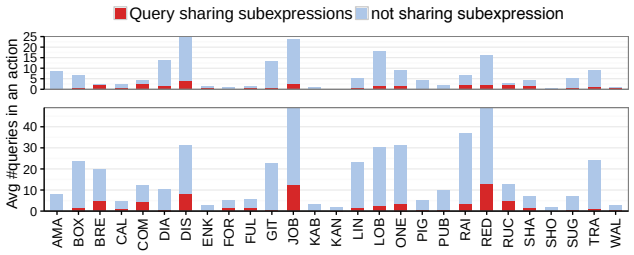

Figure 5: Queries that share common subexpressions. The upper figure shows the number of queries issued in loops, and the lower shows queries not issued in loops. We separately show these two types since queries in loop are repetitively issued and thus have greater impact on performance.

Observation: 17% queries share subexpressions with other queries with most of them performance critical.
Implication: Query execution time can be reduced by caching the intermediate result of previous queries with shared subexpressions.

To simulate the effect of caching the intermediate results in Listing 1, we create a view (Q3 in Figure 4) to store the results of the common subexpression (i.e., the ordered projects with certain status) and change the queries to use the view instead (Q4 and Q5). By using the cached results, the total query execution time of Q1 and Q2 is significantly reduced from 3.6 to 1.2 seconds (67%).

There has been prior work on identifying shared subexpressions in the context of multi-query optimization by batching and analyzing queries online [25–27]. This imposes a performance penalty on all queries. Instead, using static analysis offline incurs no runtime overhead and is still able to find many queries that potentially share subexpressions. Static analysis alone may result in false positives: if a query shares subexpressions with queries in different branches, such analysis may propose a strategy to cache all subexpressions but at runtime only one branch is taken and one subexpression will be useful. While this brings extra caching overhead, our manual check on all applications finds that very few such cases arise.

4.1.2 Fusing queries. Checking the query logs reveals that the results of queries are often only used to issue subsequent queries. Listing 2 shows an example of such queries. Q1 in the original implementation returns all members from group 1, with the results (m) only used to issue a subsequent query Q2 to retrieve the corresponding issues. Each query incurs a network round trip between the DBMS and application server and application server will serialize query results into objects after the results returned. Combining such queries can reduce the amount of data to be transferred, reducing the time spent on network and serializing data.

```
Q1: m = SELECT * FROM members WHERE group_id = 1;
Q2: SELECT * FROM issues
WHERE creator_id IN (m.id) AND is_public = 1;
```

Listing 2: Original queries (abridge from redmine [14]) listing issues by members from group 1.

We use static analysis on the AFG to identify queries whose results are only used to issue subsequent queries, with the goal to fuse them such that their results do not need to return to the application. To understand how query results are used, we trace the dataflow from each query node in the AFG until we reach either a query function node, or a node having no outgoing dataflow edge. Such read query sinks can be classified as: (1) query parameters used in subsequent queries; (2) results rendered in the view; (3) values used in branch conditions; or (4) values assigned to global variables. After analyzing the sinks, we count the number of queries that only have sinks belonging to (1), i.e., they are queries that can be fused. The result is shown in Figure 6.

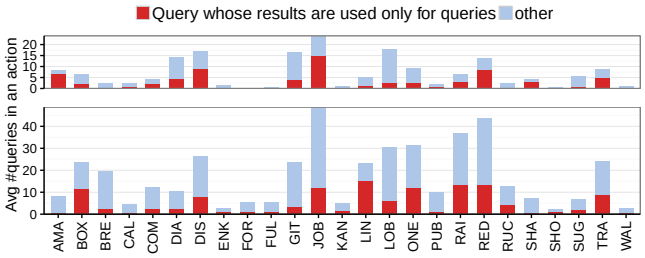

Figure 6: Queries in loop (top) and not in loop (bottom) whose results are only used in issuing further queries.

Observation: 33% of queries return results that are only used to issue subsequent queries.
Implication: Merging such queries into subsequent queries as subqueries or join queries can avoid unnecessary result transfer. Static analysis can be used to identify and rewrite such queries.

Fusing queries can lead to significant performance gain. In the previous example, Q1 and Q2 can be combined into a single query as shown in Listing 3. This reduces execution time from 1.46 and 1.3 seconds for executing Q1 and Q2 respectively to 1.02 seconds for executing the fused query (reduction of 62.3%). Moreover, fusing Q1 and Q2 avoids returning the result of Q1 (20K records, 340KB in

size in our experiments) to the application server, which brings further performance gain due to less data transfer over network and reduced serialization effort.

```
SELECT * FROM issues INNER JOIN members ON
members.group_id = 1 AND issues.is_public = 1
issues.creator_id = members.id
```

Listing 3: Combining Q1 and Q2 in Listing 2

However, such optimization can also lead to a few issues. First, if a query's result is used as a parameter to more than one subsequent queries, then query fusion will lead to repeated query execution. Fortunately, using common subexpression optimization discussed in Section 4.1.1 can avoid repeated work. Secondly, the performance of combined queries is dependent on the query optimizer, so combined query may not be always faster than the original separated queries.

4.1.3 Eliminating redundant data retrieval. We find that many queries issued by the applications retrieve fields that are not used in subsequent computation. By default, query functions provided by ORMs fetch entire rows (i.e., `SELECT *`) from the database, unless programmers use explicit functions to project specific columns from the table (e.g., using `select` in Rails). Unfortunately, such project functions are rarely used as programmers who write model functions to retrieve objects are usually unaware of how their functions will be subsequently used (possibly due to structuring the application using Model-View-Control [30]). As such, automatically identifying and avoiding unnecessary data retrieval can reduce both query execution time and amount of data transferred.

To do so, we use the AFG to identify the fields retrieved by each query along with their subsequent uses. Next, we calculate the amount of unused data. For each fixed-size field like integers, we use the data size stored in the database; for unbounded size field (e.g., `varchar`) we use 2000 bytes.[2] The average sizes of used and unused fields are shown in Figure 7.

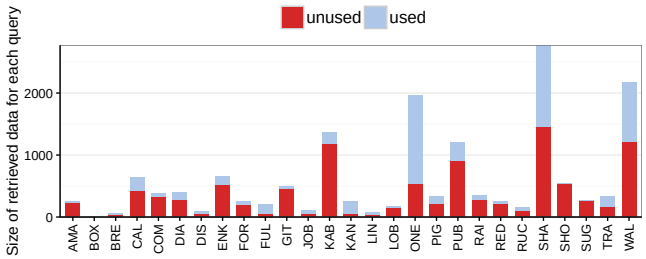

Figure 7: Amount of unnecessary data retrieved but not subsequently used in the application.

Observation: More than 63% of all retrieved data is not used in the application after retrieval.
Implication: Data transferred can be significantly reduced by not retrieving unused data. This can be achieved using static program analysis to identify the unused data and rewrite queries.

After identifying such fields, queries can be rewritten such that only used columns are retrieved using projection as mentioned above. Prior work has studied the unnecessary column retrieval problem [19] but did not propose an effective way to automatically identify them. In particular, it only evaluates the performance impact by analyzing the program and the query log obtained from dynamic profiling. Our method shows that using only static analysis on AFGs can effectively detect both retrieved and used columns, and rewrite queries automatically to avoid redundant data retrieval.

[2] [4] and [2] conclude that the most popular length of a comment is around 200 words and in average each word is 10 bytes.

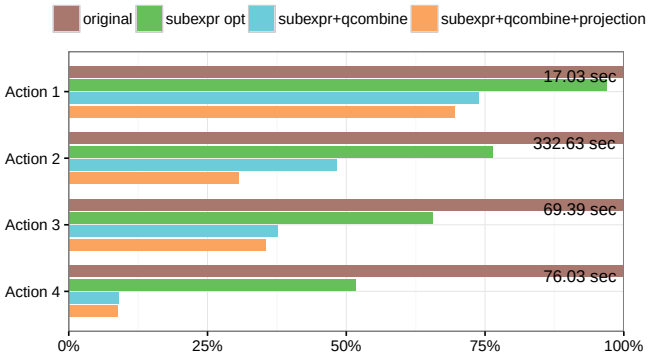

Figure 8: Performance gain after applying all optimizations

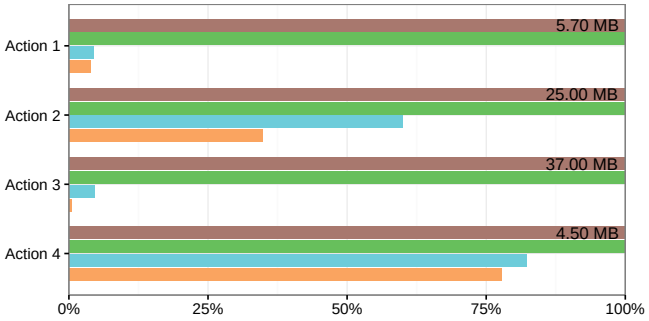

Figure 9: Transfer size reduction

4.1.4 Combining optimizations. We choose four most query intensive actions in the redmine [14] application to evaluate the optimizations mentioned above. For each action, we evaluate both the query time and the size of data transfered from DBMS. Figure 8 and Figure 9 show the results. Despite the overhead described in Section 4.1.1 and 4.1.2, each optimization still improves the overall performance. Adding up all optimizations significantly reduces the query time, up to 91%. For transfer size, the most reduction comes from fusing queries and eliminating redundant data retrieval. Transferred data in three actions is reduced by more than 60%. These results show the significance of the inefficiencies that we have identified and the potential performance gain that can be be obtained.

4.2 Rendering query results

After examining the queries that are issued by the applications, in this section we analyze how the application server processes the query results and renders them.

We observe that loops are usually the cause of performance inefficiencies in the processing of query results. By analyzing loops in the AFG, we find that 99% of them iterate over arrays or maps; 49% process results from queries issued in the current action, while

the remaining 51% iterate over user inputs or query results from previous actions. For example, when a user labels all messages as read on a webpage and clicks the "submit" button, the list of messages (which are query results from a previous action) are sent to the current action to be processed iteratively.

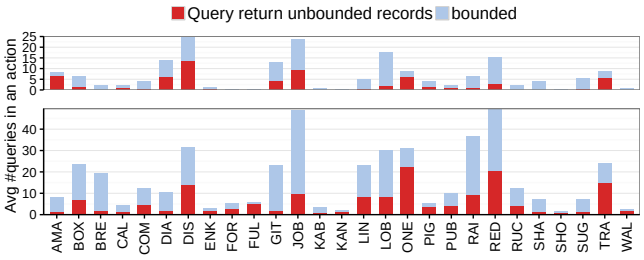

Figure 10: Queries in loop (top) and not in loop (bottom) returning bounded/unbounded # of records

This observation suggests that if a query returns a large number of tuples, then the controller or page renderer will likely be slow due to individual (rather than batched) tuple processing. Such queries also bring scalability issues: as the database size increases, the time spent on processing or rendering the query result may increase linearly or even superlinearly, making the application unable to scale well with its data. To quantify this, we analyze the result size of each query. Specifically, we check whether each query returns an increasing number of tuples with increasing database size.

To do so, we first need to estimate the size of the query results. In Rails, a query can return a larger amount of results when the database contains more tuples (we call it an "unbounded result") in all but the following cases: (1) the query always returns a single value (e.g., a `COUNT` query); (2) the query always returns a single record (e.g., retrieving using a unique identifier); (3) the query uses a `LIMIT` keyword bounding the number of returned records. Our static analyzer examines all queries in each application and determine whether a query returns bounded or unbounded result based on the query type discussed above. We then count the average number of both queries, with the result shown in Figure 10.

Observation: 36% of queries return unbounded numbers of records.
Implication: Such queries are likely to be the scaling bottleneck and can be identified by static analysis.

Turning queries from returning unbounded to bounded results often requires changing the application logic. Pagination and incremental loading are common techniques to bound the amount of data shown on a webpage. For instance, developers can change an application to to display messages over a number of pages rather than a single one. This allows the messages to be incrementally loaded as the user scrolls down the page. We manually apply pagination to three pages from three different applications, where these pages are the most rendering-time consuming pages in their respective application. We evaluate the rendering time before and after pagination, with the results shown in Figure 11.

Our evaluation shows that pagination provides impressive performance gains, reducing rendering time by 85%. As such, building tools that can identify such queries and suggest possible code changes will be an interesting area for future research.

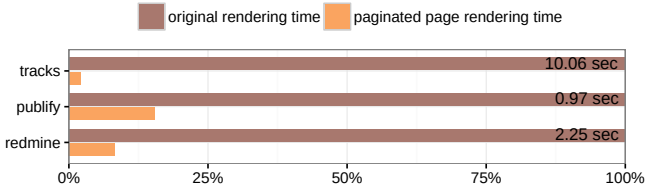

Figure 11: Pagination evaluation. The data size is chosen according to Section 3.3. The original page renders 1K to 5K records, as opposed to 40 records per page after pagination.

5 MULTI-ACTION OPTIMIZATIONS

This section presents our findings on performance issues and potential optimizations beyond individual actions.

5.1 Caching

By default, Rails does not maintain state after an action returns (i.e., the resulting webpage has been generated). Although Rails provide APIs for sharing states across actions (e.g., caching APIs that store fragmented pages or query results), using them complicates program logic and introduces user-defined caches that need to be maintained. Unfortunately, not caching query results often leads to redundant computation being performed across multiple actions.

We analyze our chosen applications and find two query patterns that can benefit from cross-action caching. Below we first introduce these two patterns, and then discuss how AFGs can be used to automatically identify and quantitatively measure how common these patterns are. Finally, we manually implement caches to evaluate the potential performance benefits of cross-action caching.

Syntactically equivalent queries across actions: We find that common practices in Rails applications can cause many syntactically equivalent queries to be issued across actions. First, Rails support filters. A filter f of a class C is executed every time a member function of C is invoked. Consequently, the same queries in f are issued by many different actions as they invoke functions of C. Checking user permissions represents one such type of filters that is shared across many actions. Second, many pages share the same partial layout. Consequently, the same queries are repeatedly issued to generate the same page layout in different actions. For example, a forum application shows the count of posts written by a user on almost every page after the user logs in.

This pattern reveals an optimization opportunity — we can identify queries that will probably get issued again by later actions, and cache their results to speed up later actions, assuming that the database contents have not been altered.

Queries with the same template across actions: We observe that many queries with the same template, i.e., queries with equivalent structures but with different parameters, are issued across actions. One major reason for this is *pagination*, a widely used programming practice to reduce rendering time as discussed in Section 4.2. As a user visits these pages, the same actions with different parameters, such as page ID, are repeatedly invoked, thus issuing queries with the same template (e.g., the ones shown in Listing 4).

This pattern reveals an opportunity similar to common subexpression optimization. For example, if the sorted posts computed when processing `Q1` are cached (i.e., the query that corresponds to

`Post.order('created')`), then `Q2` can simply return the next batch of posts from the ordered list.

```
Q1: SELECT * FROM posts ORDER BY created LIMIT 40
    OFFSET 0
Q2: SELECT * FROM posts ORDER BY created LIMIT 40
    OFFSET 40
```

Listing 4: Q1 and Q2 are issued when visiting page1 and page2, sharing the same query template

We apply static analysis on the AFG to quantitatively understand how common the above two patterns are across different applications. Specifically, we analyze every *previous-current* action pair that is linked by the *next action* edge in the AFG described in Section 3. For each query `Q` in the *current* action, we check if there exists a query `Q'` from the corresponding *previous* action that is generated by the same code (e.g., the same filter or the same function) as `Q`. If such a `Q'` exists, `Q` and `Q'` share the same query template. We further examine their parameters to see whether they are syntactically-equivalent queries — if `Q` only takes constant value or the same data from the session cache as parameters, we consider it to be syntax-equivalent to `Q'` (i.e., same template and same parameter)(①); if `Q` takes user input and/or utility function result as parameter, we consider it to be template-equivalent to `Q'` with potentially different parameters(②).

We calculate the average number of the two types of queries (① and ②) issued in an action. The static analysis result is shown in Figure 12. If a query has a syntax or template-equivalent peer in any previous action, query processing can use the cached result.[3]

Observation: 20% and 31% of the queries are syntactically or template equivalent to a query in a previous action respectively. **Implication:** A cross action query cache can be used to accelerate query execution. Program analysis can be used to accurately identify such queries and determine which result to be cached.

We evaluate the benefit of caching using sugar [16], a discussion forum application. Since it is difficult to predict a user's complete page-access sequence, our evaluation focuses on a common visit pattern on paginated webpages: after visiting the first page, users often visit the following pages to see subsequent results. We choose four slowest paginated webpages that show 1) latest discussions; 2) popular discussions; 3) latest posts; and 4) recent invitations. We populate the database following the data distribution of online forum website as mentioned in Section 3.3, with 6GB data in total. Each page takes 0.2 to 24.3 seconds to generate, with more than 95% of time spent on database queries. We measure the total query time for each page since our caching optimizations only affect queries.

We then use the AFG to automatically identify which query result to cache, manually cache the results, and measure how much query time is saved. The results are shown in Figure 13. Although the query time in the first page becomes slightly longer (at most 5%) due to caching temporary results, queries in following pages are significantly faster, by more than 5× in all actions (up to 245×). The results illustrate an impressive benefit of caching to improve application performance.

[3]We imagine such cache can be invalidated similar to standard application caches [17].

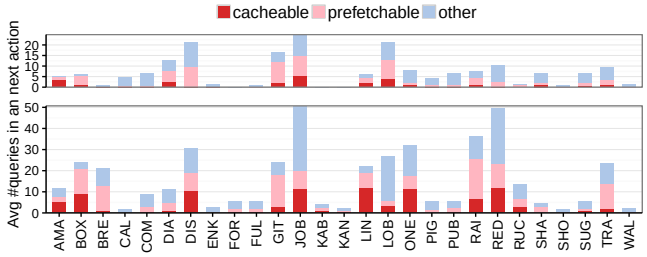

Figure 12: Cacheable and prefetchable queries in loop(top) and not in loop(bottom)

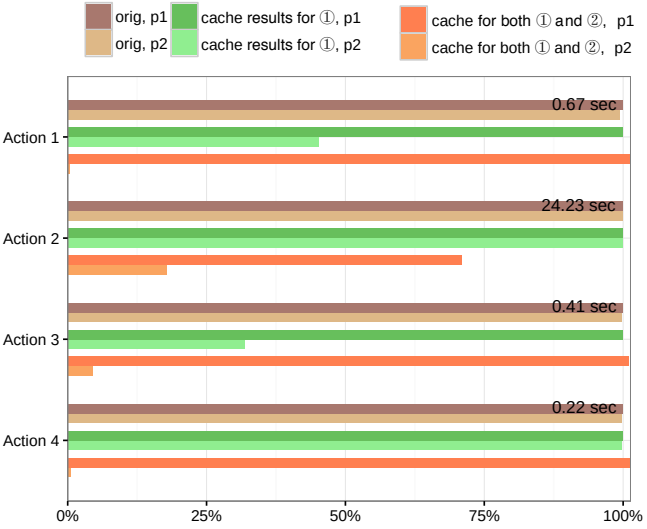

Figure 13: Caching evaluation on paginated webpages. p1 and p2 refer to the current the next pages, respectively. (we observed no significant latency difference among the pages that are reachable from p1). The x-axis shows the query time percentage with original first page's query time as basline.

5.2 Storing data on the disk

In this section, we examine optimizations beyond action flow graphs. More specifically, we present our findings related to how persistent data is organized (i.e., the physical design) on the DBMS.

Our analysis finds that the poor default physical design produced by ORM frameworks is one of the major causes for performance inefficiencies. We observe that many queries are programmatically generated with a few parameters provided by users during runtime, and many queries only use a subset of the object fields that are persistently stored. Such queries can be partially evaluated with the results stored using customized physical layout, and query evaluation can work on much less data using customized layout. In the following we discuss different cases where such situation arises. Then we evaluate the prevalence of each case and suggest potential optimization opportunities.

5.2.1 Partial evaluation of selections. Many selection predicates of programmatically generated queries use constant values as parameters. For instance, a page retrieving all commit actions issues a query with the predicate `name='COMMIT'` on the `actions` table, where

'COMMIT' is a constant; a page showing all unexpired stories issues a query with `expired=0` on the `stories` table as shown in Listing 5, where again `0` is a constant. In these cases, we can partially evaluate the query with known parameters, and store the results such that only the remaining (user input dependent) portion of the query is evaluated at runtime.

```
SELECT s.id, s.votes FROM stories AS s
WHERE s.expired=0 AND s.votes>0 AND s.created>?
INNER JOIN tags AS t ON t.sid=s.id AND t.tagid=?
ORDER BY s.created DESC
```

Listing 5: Query abridged from lobsters [12] that retrieves ID and votes of unexpired recent stories with positive votes and a certain tag.

```
SELECT s.id, s.votes FROM stories AS s
WHERE s.created>?
INNER JOIN tags AS t ON t.sid=s.id AND t.tagid=?
ORDER BY s.created DESC
```

Listing 6: Query retrieving stories on row partitioned table, with the predicates `s.expired=0` and `s.votes>0` being partially evaluated.

We statically analyze every predicate to see how common such constant-parameter predicates are. For every selection predicate, we locate the data sources of all its parameters following the data dependency edges in the AFG. We consider a predicate to be "constant" if the data sources of its parameters are constant values (i.e., not user inputs, query results, utility function return values, or any other non-constant sources). We check every query to see whether it contains any constant predicate; the number of such queries is shown in Figure 14.

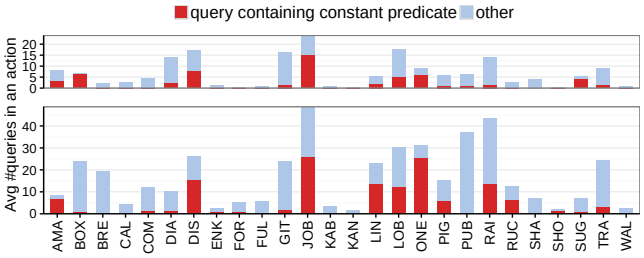

Figure 14: Average # of queries in loops (top) and not in loops (bottom) with constant predicates.

> **Observation:** 33% of the queries includes predicate with constant value as parameter.
> **Implication:** These queries can be partially evaluated to reduce query execution time.

Given a constant predicate `p` of query `Q` on table `T`, one way to partially evaluate `Q` is to partition `T` row-wise into two tables: one holding tuples satisfying `p` and another holding tuples not satisfying `p`. This can be automated by first using static program analysis to identify every such constant predicate `p`, then change the physical layout of the corresponding table `T`, and rewrite the query `Q` to execute on the partitioned table. If there are N queries with different constant predicates on one table, the table can be partitioned recursively, each time using constant predicates from one query, into 2^N partitions. In practice, N is usually small: our static analysis shows that on average each table is split into 3.2 partitions using the partition scheme mentioned above.

Listing 5 shows a query with constant predicates from the lobsters application. Based on these predicates the stories table can be partitioned into two — the first partition stores unexpired stories with positive votes (i.e., `s.expired=0 AND s.votes>0`), and the second stores all other stories. The rewritten query is shown in Listing 6 that runs only on the first partition. We measure the performance of the transformed query and find that its execution time is reduced from 2.12 to 1.67 seconds (a 21% improvement). The evaluation uses 500K `story` tuples, a setting chosen based on real-world application statistics as described in Section 3.3.

5.2.2 Partial evaluation of projections. Many queries only use a subset of all stored fields in a table. For example, a query counting the number of recent posts written by a user only uses the `user_id` and `created` fields of the `posts` table. However, since ORM frameworks map each class to a database table by default, while all fields are stored in a single table regardless of their usage patterns. Instead of only reading `user_id` and `created`, the post-counting query reads the whole tuple, retrieving much more data than needed.

Furthermore, we find many cases where larger fields (in terms of size) of a table are used by a lot fewer queries than smaller fields. For instance, many webpages display a list of short tuple summaries, where each summary contains small fields. In constrast, many large fields are only displayed on pages that show *all* stored fields. For example, a page showing a list of posts only shows the title, author, and creation time of each post, all of which are small in size; meanwhile, the entire text body is only shown on a page that renders everything pertaining to a particular post.

The above observation leads to an optimization opportunity: if we co-locate fields that are frequently used together in queries, we can speed them up by reducing the amount of data retrieved, especially when such fields are small in size. To understand the benefits of this optimization, we use static analysis on the AFG to quantitatively measure: 1) how often queries use only a subset of object fields rather than `SELECT *` (in this experiment we assume that the queries only retrieve the necessary fields as described in Section 4.1.3); and 2) how large are the fields used in these queries, compared to the unused fields. Figure 15 shows the result for 1), and we only summarize the result for 2) due to space limit.

> **Observation:** 61% of the queries use only a subset of fields. 45% of all fields (in number) are used in these queries, and these fields account for only 26% in terms of data size.
> **Implication:** Many queries can be sped up by partially evaluating projections (i.e., co-locating the used fields as described above).

One way to achieve data co-location is to vertically partition a table into two: one contains the fields used in all non-"`select *`" queries, and the other contains all other fields. Doing so can speed up many queries. For instance, in the lobsters application, after statically analyzing the field usage pattern, we manually partition the `stories` table vertically into two part, with the first partition containing only 9 out of the 19 total columns. Particularly, this partition does not include some large fields holding user-input text like `title` and `description`, and consists of only 9.5% of the data

originally stored in `stories` table. After partitioning, we rewrite the query as shown in Listing 5, which reads and processes full `story` tuples, to the one shown in Listing 7, which only retrieves the fields stored in the partitioned table. Doing so reduces query execution time by 22.6%, from 2.12 to 1.64 seconds.

```
SELECT s1.id, s1.votes, s1.uid FROM s1
WHERE expired=0 AND votes>0 AND created>?
INNER JOIN tags AS t ON t.sid=s1.id AND t.tagid=?
ORDER BY s1.created DESC
```

Listing 7: Query retrieving stories from the vertically partitioned table and returning the same results as Listing 5.

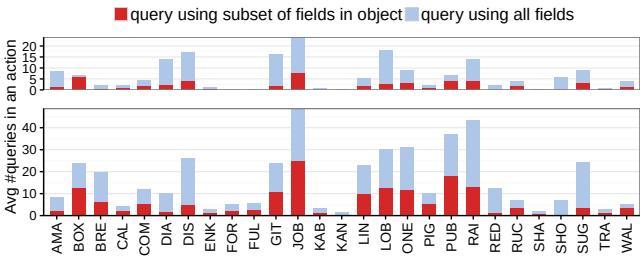

Figure 15: Number of queries in loops (top) and not in loops (bottom) that use only a subset of object fields.

While using only a subset of fields can reduce query execution time, vertical partitioning introduces overhead to queries that use all fields in an object by adding an extra join based on the object's key. It also increases write query overhead since each record is split and written into multiple tables after optimization. However, our profiling finds that the join overhead is trivial if the key on the partitioned tables is indexed, and the write overhead is small since each write query usually affects a very small number of tuples.

5.2.3 Table denormalization. In addition to selections, joins can also be partially evaluated: when the join predicates are fixed, we can store the pre-joined (i.e., denormalized) tables to reduce query time. This can lead to significant performance gain in many applications, as join queries are often computationally expensive.

To better understand the benefits of table denormalization, we use static analysis to count the number of join queries and the average number of tables involved in these join queries. The result is shown in Figure 16.

Observation: 55% of the queries are join queries, and on average each join involves 2.8 tables.
Implication: Improving join query performance is important to ORM applications. Table denormalization can reduce the execution time of join queries in many applications.

While pre-joining tables can accelerate join queries, it also has downsides. First, pre-joining can duplicate a large amount of data. Second, it slows down write queries, as well as read queries that do not use all fields in the denormalized table, especially read queries that only access one of the joined tables. Fortunately, combining vertically partitioning and pre-joining can avoid duplicating too much data or slowing down non-join read queries. Only the fields used in the join query are denormalized to be stored in one table while the other fields remain in the original table. Our evaluation result

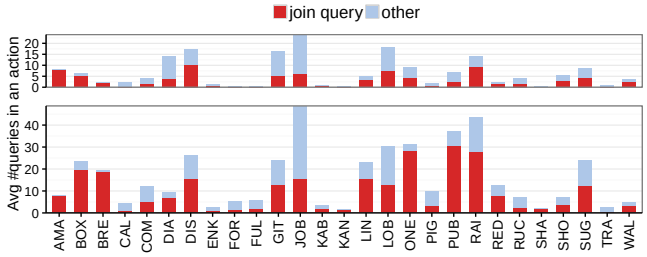

Figure 16: Average number of join queries in loop (top) and not in loop (bottom).

(described later in Figure 17) shows that the combined optimization achieves large performance improvement.

5.2.4 Combining optimizations. In this section we evaluate the gain of optimizations introduced in previous sections. We choose four actions that answer GET requests, and three actions that answer POST requests. A GET request is generated when a user visits a webpage, which mostly issues read-only queries. A POST request is sent when a user submits a form, which issues both read queries to retrieve relevant data (e.g., data for authentication) and write queries to record the user data and action. We choose both the GET and POST actions based on the query time: the chosen actions are those spending most time on database queries among GET/-POST actions correspondingly. We then apply each optimization discussed earlier one at a time to all queries in these actions, and evaluate the query time for each action.

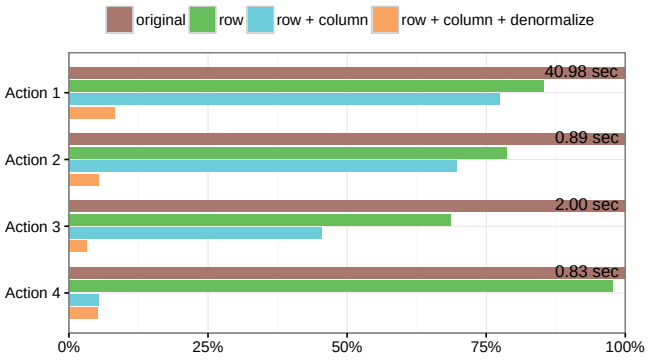

Figure 17: Performance of the original and optimized queries from GET actions

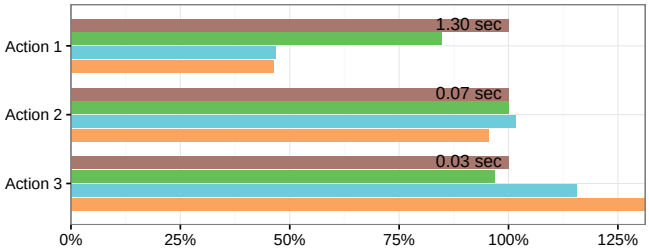

Figure 18: Performance of the original and optimized queries from POST actions

Figure 17 and Figure 18 show how query time changes under different optimizations. For GET actions, each optimization reduces

the query time and combined optimization improves query performance by up to 41×. For POST actions that include both read and write queries, the gain varies across actions. Some optimizations slow average performance due their overhead on write queries (as explained previously in each optimization).

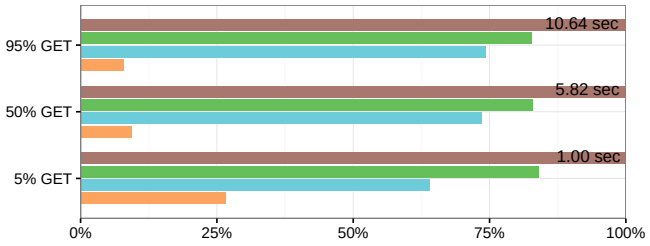

Figure 19: Performance of original and accumulated optimizations for a mixed of GET/POST.

Despite the overhead, in most cases overall query performance is largely improved due to the great improvement in GET actions, which are usually the application bottleneck. The write queries usually change a small number of records and are much faster than long-running read queries, so the overhead on writes appears to be trivial. Figure 19 shows the average query time under different mixes of GET/POST actions. When 5% of actions are POST, the combined optimization reduces the query time by 92.2%. Even when 95% of actions are POST, the combined optimization still reduces 73.4% of query time, showing a great benefit of our proposed physical-design optimizations.

6 RELATED WORK

In addition to the prior work mentioned in Section 4.1, we now discuss two additional categories of related work.

Empirical Studies. A previous study [18] investigated performance anti-patterns for ORM applications. However, it only mentioned two anti-patterns from three Hibernate-based applications. We believe such anti-patterns are addressed by previous work [21, 29]. We instead provide a thorough study of nine patterns of query-related performance problems on a larger range of applications, and discuss solutions that were not covered in prior work.

Program analysis for database optimization. Our optimizations share the same high-level idea with recently proposed techniques. DBridge [24] includes a series of work on holistic optimization. This series of work includes query batching and binding, automatic transforming of regular object-oriented code into synthesized queries, decorrelation of user functions and queries, etc. Other holistic optimizations include but not limited to: StatusQuo [20, 22], Sloth [21] and QBS [23] for query synthesis, QURO [31] for query reordering in transactions, PipeGen [28] for automatic data pipe generation, etc. Our work instead proposes a number of new observations and research opportunities that can leverage prior proposed techniques.

7 CONCLUSION

In this paper, we studied the database-related inefficiencies in 27 real-world web applications that are built using the Rails ORM framework. We built a static analyzer to examine how these applications interact with databases through the ORM. We also profiled some applications using workloads that follow real-world data distributions. Our findings reveal many optimization opportunities and research challenges for designing performance-efficient ORM frameworks in the future.

8 ACKNOWLEDGEMENTS

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

REFERENCES

[1] Active record. https://github.com/rails/rails/tree/master/activerecord.
[2] Common length of a comment. https://www.reddit.com/r/dataisbeautiful/comments/2mkp6u/comment_length_by_subreddit_oc/.
[3] diaspora. https://diasporafoundation.org/.
[4] Discussion on blog length. https://www.snapagency.com/blog/whatll-best-length-blog-article-2015-seo/.
[5] Django. https://www.djangoproject.com/.
[6] Enterprise javabeans. https://jcp.org/en/jsr/detail?id=220.
[7] filter, validation and association. http://guides.rubyonrails.org.
[8] gitlab. https://github.com/gitlabhq/gitlabhq.
[9] gitlab forum. https://gitlab.com/gitlab-org/gitlab-ce/issues.
[10] Hibernate. http://hibernate.org/.
[11] kandan:web. http://getkandan.com/.
[12] lobsters. https://github.com/jcs/lobsters.
[13] publify. https://github.com/publify/publify.
[14] redmine. https://github.com/redmine/redmine.
[15] Ruby on rails. http://rubyonrails.org/.
[16] sugar. https://github.com/elektronaut/sugar.
[17] Update query cache. http://instantbadger.blogspot.com/2009/12/memcached-cache-invalidation-made-easy.html.
[18] 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 *ICSE*, pages 1001–1012, 2014.
[19] 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 Transactions on Software Engineering*, 2016.
[20] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. Myers. Statusquo: Making familiar abstractions perform using program analysis. In *CIDR*, 2013.
[21] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). In *SIGMOD*, pages 931–942, 2014.
[22] A. Cheung, S. Madden, A. Solar-Lezama, O. Arden, and A. C. Myers. Using program analysis to improve database applications. *IEEE Data Engineering Bulletin*, 37(1):48–59, 2014.
[23] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In *PLDI*, pages 3–14, 2013.
[24] K. V. Emani, K. Ramachandra, S. Bhattacharya, and S. Sudarshan. Extracting equivalent sql from imperative code in database applications. In *SIGMOD*, pages 1781–1796, 2016.
[25] S. Finkelstein. Common expression analysis in database applications. In *SIGMOD*, pages 235–245, 1982.
[26] G. Giannikis, G. Alonso, and D. Kossmann. Shareddb: Killing one thousand queries with one stone. In *VLDB*, pages 526–537, 2012.
[27] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *SIGMOD*, pages 383–394, 2005.
[28] B. Haynes, A. Cheung, and M. Balazinska. PipeGen: Data pipe generator for hybrid analytics. In *SOCC*, pages 470–483, 2016.
[29] K. Ramachandra, M. Chavan, R. Guravannavar, and S. Sudarshan. Program transformations for asynchronous and batched query submission. *IEEE Transactions on Knowledge and Data Engineering*, pages 531–544, 2015.
[30] M. C. Selfa, Diana M. and M. D. R. Boone. A database and web application based on mvc architecture. In *CONIELECOMP*, 2006.
[31] C. Yan and A. Cheung. Leveraging lock contention to improve OLTP application performance. In *VLDB*, pages 444–455, 2016.