REFORMULATOR: Automated Refactoring of the N+1 Problem in Database-Backed Applications

Alexi Turcotte
Northeastern University
Boston, MA, USA
turcotte.al@northeastern.edu

Mark W. Aldrich
Tufts University
Medford, MA, USA
mark.aldrich@tufts.edu

Frank Tip
Northeastern University
Boston, MA, USA
f.tip@northeastern.edu

ABSTRACT

An Object-Relational Mapping (ORM) provides an object-oriented interface to a database and facilitates the development of database-backed applications. In an ORM, programmers do not need to write queries in a separate query language such as SQL, they instead write ordinary method calls that are mapped by the ORM to database queries. This added layer of abstraction hides the significant performance cost of database operations, and misuse of ORMs can lead to far more queries being generated than necessary. Of particular concern is the infamous “N+1 problem”, where an initial query yields N results that are used to issue N subsequent queries. This anti-pattern is prevalent in applications that use ORMs, as it is natural to iterate over collections in object-oriented languages. However, iterating over data that originates from a database and calling an ORM method in each iteration may result in suboptimal performance. In such cases, it is often possible to reduce the number of round-trips to the database by issuing a single, larger query that fetches all desired results at once.

We propose an approach for automatically refactoring applications that use ORMs to eliminate instances of the “N+1 problem”, which relies on static analysis to detect data flow between ORM API calls. We implement this approach in a tool called REFORMULATOR, targeting the Sequelize ORM in JavaScript, and evaluate it on 8 JavaScript projects. We found 44 N+1 query pairs in these projects, and REFORMULATOR refactored all of them successfully, resulting in improved performance (up to 7.67x) while preserving program behavior. Further experiments demonstrate that the relative performance improvements grew as the database size was increased (up to 38.58x), and that front-end page load times were improved.

KEYWORDS

databases, ORMs, program analysis, refactoring, JavaScript

1 INTRODUCTION

An ORM (Object-Relational Mapping) provides an object-oriented facade for a database enabling programmers to access it using ordinary method calls. The ORM maps such method calls to database queries and converts query results to objects in the host language so that programmers do not need to use a separate database query language like SQL to interact with the database. However, the added layer of abstraction introduced by ORMs may obscure the cost of database operations, and careless ORM usage may generate more database queries than are necessary, causing poor performance.

Of particular concern is the infamous “N+1 problem” [9, 12, 38], which arises when an initial database query yields N results that are then used to issue N subsequent database queries. This can lead to significant performance problems because database queries are typically high-latency operations. The “N+1 problem” anti-pattern frequently occurs in applications that use ORMs, where it often arises in the following scenario:

- An initial call to the ORM’s Application Programming Interface (API) generates a database query that results in a collection C of objects.
- Then, a loop iterates through C and, for each element $c \in C$, calls an ORM API method with c as an argument, resulting in the generation of another new database query.

We found that, in many of these cases, the “N+1 problem” can be remediated by inserting a single ORM API call that has the effect of retrieving the information from the database that was previously fetched by the N subsequent queries. This refactoring, by significantly reducing the number of round-trips to the database, can drastically improve performance.

We present an approach for automatically detecting instances of the “N+1 problem” and generating code transformations that reduce the number of database queries. To detect instances of the “N+1 problem”, a static data-flow analysis detects data flow from the result of one ORM API call to an argument of another ORM API call, where the latter call occurs within a loop. To repair these instances, we define a set of declarative rewrite rules that specify how code should be transformed to reduce the number of generated queries. These transformations result in code that: (i) issues a constant number of queries, (ii) is behaviorally equivalent, and, importantly, (iii) performs better and scales as database size increases.

We implement this technique in a tool called REFORMULATOR, targeting the Sequelize ORM for the JavaScript programming language, and evaluate it on 8 JavaScript projects that use Sequelize. In these projects, REFORMULATOR found 44 instances of the “N+1 problem”. Due to the highly dynamic nature of the JavaScript programming language, sound static analysis for JavaScript remains elusive [20, 21, 27], and as a result, it is possible for our implementation to propose refactorings that do not preserve behavior. Therefore, following other recent work on refactoring for JavaScript [6, 15], REFORMULATOR presents refactorings as *suggestions* that should be carefully vetted by a programmer, e.g., by running tests.

Conference’17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In practice, REFORMULATOR successfully refactored all 44 instances of the "N+1 problem", and in all cases performance was improved (up to 7.67x, even with small amounts of data being processed). Additional experiments revealed speedups of up to 38.58x and substantial improvements in scalability by demonstrating that the relative performance improvements grew as the database size was increased. We also confirmed that these performance gains translate to an improved user experience, by demonstrating reductions in page load times by up to 90% with large database sizes.

In summary, the contributions of this paper are:

- An approach in which instances of the "N+1 problem" are detected by tracking data flow between ORM API calls, and where a set of declarative rewrite rules specifies how code can be refactored to eliminate them;
- An implementation of this approach in a tool called REFORMULATOR, targeting the popular Sequelize ORM in JavaScript;
- An evaluation of REFORMULATOR on 8 projects containing 44 instances of the "N+1 problem", demonstrating that the suggested refactorings improve performance and scalability, while preserving program behavior in all cases,

An artifact complete with the source code and the ability to re-run the experiments discussed in this paper is available [33].

The remainder of this paper is organized as follows. § 2 establishes relevant background via motivating example; § 3 details the approach to finding and refactoring "N+1 problem" anti-patterns; § 4 describes the implementation of this approach in a tool called REFORMULATOR; § 5 presents an evaluation of REFORMULATOR; § 6 identifies some threats to the validity of our approach; § 7 sketches the landscape of related work; and finally, § 8 concludes.

2 BACKGROUND AND MOTIVATION

To illustrate how "N+1 problem" issues arise in practice, consider **youtubeclone** [23], a popular open source video-sharing application emulating YouTube with over 125 stars and nearly 600 forks.

Like many database-backed web applications, the three components of **youtubeclone** are a front-end client-side interface, a back-end server, and a database. As users navigate through the front-end, HTTP requests are made to the server which sends HTTP responses once the requests have been processed. In some cases, the server will query the database if data is needed to prepare the response.

youtubeclone is written in JavaScript, and uses Sequelize [3], a popular ORM that enables JavaScript applications to interact with relational databases. The database backing **youtubeclone** has tables for videos, users, subscriptions, and views, and Figure 1 shows the Sequelize code modeling the video and user tables (simplified for brevity). The model corresponding to the video table is defined on lines 1-12, with the primary key "vid" defined on lines 2-7, and the model corresponding to the user table is defined on lines 13-24, with the primary key "uid" defined on lines 14-19. The association between the two models is made using a *foreign key*, i.e., a table column that contains the primary key of another table. Line 26 specifies "uploader" as a foreign key into the video table. This foreign key allows *joins* to be executed on the video and user tables, which fetches the user information associated with a video. E.g., a list of videos with "ASE 2022" in the title and information related to the uploader is obtained by the following Sequelize API call:

```
1  const Video = sequelize.define("Video", {
2    vid: {
3      type: DataTypes.UUID,
4      allowNull: false,
5      primaryKey: true,
6      defaultValue: Sequelize.UUIDV4,
7    },
8    title: {
9      type: DataTypes.STRING,
10     allowNull: false,
11   },
12 });
13 const User = sequelize.define("User", {
14   uid: {
15     type: DataTypes.UUID,
16     allowNull: false,
17     primaryKey: true,
18     defaultValue: Sequelize.UUIDV4,
19   },
20   username: {
21     type: DataTypes.STRING,
22     allowNull: false,
23   },
24 });
25 // Establish association between Video and User
26 Video.belongsTo(User, {foreignKey: "uploader"});
```

Figure 1: Example database definition in Sequelize.

```
Video.findAll({include: {model:User},
  where: {[Op.substring]: { title: "ASE 2022" }}})
```

which would be translated into the following SQL query:

```
SELECT * FROM VIDEO LEFT JOIN USER ON USER.uid = VIDEO.uploader
   WHERE VIDEO.title LIKE "%ASE 2022%"
```

`Video.findAll` performs a `SELECT` from `Video` (since no attributes were specified, this is translated to `SELECT *`), `include` indicates that the generated query should include the associated `User` table by performing a `LEFT JOIN`, and `where` specifies that the query should only return videos with "ASE 2022" in the title.

SQL (and, by extension, Sequelize) also allows queries to specify a *grouping* clause, and *aggregations* over groups. If a query includes `GROUP BY ColumnName`, the results will be grouped according to unique values of `ColumnName`. Aggregate functions (such as `COUNT`) can be included in grouped queries, and the function is performed over the group. For example, the query `SELECT title, COUNT(title) FROM VIDEOS GROUP BY title` will yield all unique video titles as well as how many videos had that title.

To illustrate how ORMs may be misused, consider Figure 2(a), which shows some key fragments of a function `recommendChannels` from the back-end of **youtubeclone**. The function takes a parameter `req` representing a user request, and eventually produces an HTTP response that includes other channels that the current user (identified by `req.id`) might be interested in. This function first executes a call to `User.findAll` on lines 30–34 to determine a set of up to 10 channels for which the `id` is not the same as the current user (i.e., the current user does not own the channel). This call is mapped by the ORM to a SQL query of the form `SELECT ··· FROM User LIMIT 10`.

Later, execution enters a loop (lines 35–44) that executes a call `Subscription.findOne(···)` to determine, for each of these channels, if the current user is already subscribed to it. Each of these calls is mapped by the ORM to an SQL query that looks as follows:

```
29 async function recommendChannels(req, res) {
30   const channels = await User.findAll ({
31     limit : 10,
32     attributes : ["id", "username", "avatar", "channelDescription"],
33     where: { id: { [Op.not]: req.user.id } }
34   });
35   channels.forEach(async (channel, index) => {
36     const isSubscribed = await Subscription.findOne({
37       where: {
38         subscriber : req.user.id,
39         subscribeTo : channel.id,
40       },
41     });
42     channel.setDataValue("isSubscribed", !! isSubscribed);
43     // send HTTP response after processing the last channel
44   });
45 }
```

```
46 async function recommendChannels(req, res) {
47   const channels = await User.findAll ({
48     limit : 10,
49     attributes : ["id", "username", "avatar", "channelDescription"],
50     where: { id: { [Op.not]: req.user.id } }
51   });
52   const subscriptions = await Subscription.findAll ({
53     where: {
54       subscriber : req.user.id,
55       subscribeTo : channels.map(chan => chan.id)
56     }
57   });
58   channels.forEach(async (channel, index) => {
59     const isSubscribed = subscriptions.find(data => data.subscribeTo === channel.id);
60     channel.setDataValue("isSubscribed", !! isSubscribed);
61     // send HTTP response after processing the last channel
62   })
63 }
```

Figure 2: (a) Functionality for recommending channels in *Youtube Clone*, exhibiting the "select N+1 problem". (b) Refactored version of the code, which generates fewer SQL queries.

`SELECT ··· FROM Subscription WHERE (Subscription.subscriber = ··· AND Subscription.subcribeTo = ···) LIMIT 1`. In other words, an initial query creates N results (here, $N = 10$) and *subsequently, a query is issued for each of these N results*, requiring a total of $N + 1$ database round-trips. The ORM community has recognized that, in such situations (referred to as the "N+1 problem"), it is often possible to modify the code to issue a lower, constant number of queries.

Figure 2(b) shows how the code of Figure 2(a) can be refactored to accomplish this. Here, an additional query is added on lines 52–57 to obtain an array `subscriptions` containing the channels from `channels` that the current user is subscribed to; on line 55, the `channels.map(...)` retrieves all of the ids for each `channel` so that the ORM can fetch the subscription status for all of the channels at once. In addition, in the loop over all channels (lines 58–62), the subscription status for a given channel is now looked up by calling the standard `find` method on arrays instead of querying the database. As a result, only 2 SQL queries are needed instead of the original N+1 queries.

`recommendChannels` contains two additional instances of the "N+1 problem" and both could be refactored similarly. The refactored code outperforms the original by a factor of nearly 3x.

Note the `await` on line 30: calls to Sequelize are asynchronous operations implemented using *promises* [1]. A promise is an object that represents the value computed by an asynchronous computation, and an `await` expression indicates to JavaScript that it should suspend execution of the current function until the asynchronous computation being `await`-ed is completed. If the computation resulted in a value v, we say that the promise p associated with it was resolved with v, and `await` p will return v (i.e., the value is unpacked from the promise). If the computation resulted in an error, that error will be thrown and can be caught in a try-catch. Tying this back to the example, `await User.findAll(...)` will asynchronously execute the query, and execution will resume once the data is returned.

The next section presents how the refactoring opportunities discussed in this section can be detected, and how code transformations can be generated automatically.

3 APPROACH

Our technique for suggesting refactorings that have the effect of eliminating the "N+1 problem" has two components:

(1) a data flow analysis to locate pairs of ORM API calls involved in an "N+1 problem", discussed in § 3.1, and
(2) a set of declarative rewrite rules describing how pairs of N+1-related ORM API calls are transformed to eliminate the problematic pattern, discussed in § 3.2.

3.1 Data-Flow Analysis

The main question the data-flow analysis is looking to answer is: *does data-flow exist between two ORM API calls?* Put differently, for every ORM API call C, the analysis should determine the existence of data-flow between the result of a previous ORM API call and any of C's arguments. This is achieved with a *taint analysis* [17, 19, 32], where ORM API calls are defined as *sources* of taint, and ORM API call arguments are defined as *sinks*. Concretely, we rely on a standard taint analysis framework available in CodeQL [24] to detect taint flows from sources to sinks.

For example, consider the code snippet in Fig 2(a). Here, the call to `findAll` returns a promise that will be resolved with the data from the database, and that value will flow into `channels`. Thus, there exists data-flow between `findAll` and `channels` through the promise created by `findAll`. The `forEach`-loop on lines 35-44 iterates over these values, and thus there is data-flow from elements of `channels` into the `channel` callback parameter (line 35). Finally, there is data-flow from `channel` into the argument of `Subscription.findOne` through the field access `channel.id` (line 39).

In order to generate code transformations, the approach needs the *property names* that are the target of data-flow (e.g., the analysis will report that data-flow exists between `subscribeTo : channel.id` and `channels`). Thus, the analysis notes exactly which property/value pairs $p : v$ in an ORM API call object O had values v that that were the target of data-flow from the result m of a previous ORM API call; in the following section, this process is encapsulated in the function $getAllPropertiesWithDataFlow(O, m)$.

3.2 Refactoring

Code transformations are presented as a set of declarative rewrite rules that can be found in Figure 3. The anatomy of the rules is:

$$\frac{\textit{conditions}}{\texttt{(code before)} \rightsquigarrow \texttt{(code after)}}$$

FINDALL-FINDONE. This rule depicts the transformation for a flow from `findAll` through a loop into `findOne`. An example applying this rule to the code in Figure 2 follows this description.

(1) First, the list of properties (*props*) of the argument to the `findOne` call (O_2) that are the targets of data-flow from the result of a call to `findAll` (`m1s`) is obtained through the helper function *getAllPropertiesWithDataFlow*.
(2) The goal of this transformation is to insert a new ORM API call to `findAll` replacing the old call to `findOne`, and so the argument to that new call must be constructed. The idea is adapt the argument to the old call (O_2); since the new call will be placed *before* the loop, any properties in O_2 that were targets of data-flow must be updated to map directly over the result of the previous API call (`m1s`).
To achieve this, a new object O_2' is adapted from O_2 by updating all of the values of the properties in O_2 referred to by *props* to be maps over `m1s`, through the *updatePropReferences* helper function. For all properties $p : v$ in *props*, the property `f` of the model `M1` referred to by `v`, either directly in `v` itself (e.g., if `v` is of the form `x.f`) or indirectly (e.g., if `v = x.f` earlier in the code) is obtained, and v is replaced with `m1s.map(m1 => m1.f)` in O_2'
(3) As the goal of this refactoring is to replace many calls to `findOne` with a single call to `findAll`, the result `m2s` of that new call will need to be iterated over to pick out the same data that was returned by the original call to `findOne`. `m2s` contains all of the data that would have been fetched in the loop, and the idea here is to map whatever comparisons were being made in the original call to `findOne` to some new boolean expression (`BE`) that can be used to pick out the datum of interest from the array of results (`m2s`). This is achieved through the *createArrayLookup* helper function: for each property/value pair $p : v$ in *props*, a boolean expression `m1.p === v` is added to `BE` (here, `m1` is the parameter name of a callback that will be inserted by the transformation). In constructing `BE` in this manner, the same comparisons that were being made in the old `findOne` are performed in `BE`.
(4) To enact the transformation, a fresh variable `m2s` is declared and set to the return value of a new call to `M2.findAll(`O_2'`)`, and is placed immediately before the loop; the old call to `M2.findOne(`O_2`)` is replaced with a lookup over the `m2s` array, and the entry matching `BE` is picked out.

FINDALL-FINDONE *(Walk-through).* To help illustrate the rewrite rule, consider the transformation in Figure 2.

(1) First, there exists data-flow between `channels` and the argument to `Subscription.findOne` in the `subscribeTo: channel.id` property; mapping to the FINDALL-FINDONE rewrite rule, this property will be the sole element of *props*.
(2) The new ORM API call object (lines 52-57) is obtained from the existing call object (lines 36-41), where the value of the property with data flow (`subscribeTo: channel.id`) is updated to map over `channels` (`channels.map(chan => chan.id)`; this is O_2'.
(3) A new boolean expression `BE` is built from the properties that had data from `channels` flow into them, in this case the sole property with data flow `subscribeTo: channel.id` populates `BE` with the boolean expression `data.subscribeTo === channel.id`.
(4) Putting it all together: the new call to `Subscription.findAll` is placed before the loop (lines 52-57), and the old call to `Subscription.findOne` is replaced with a `find` over the array of subscriptions returned by `Subscription.findAll` (line 59).

FINDALL-COUNT. This rule depicts the transformation for data-flow into a call to `count`. The list of properties with data flow from `m1s` is obtained with *getAllPropertiesWithDataFlow* as in FINDALL-FINDONE. The new ORM API call object O_2' is created in much the same way as well, except that in this case grouping and aggregation is added to O_2': each property name referred to in *props* is added to a grouping clause in O_2', and also to a count aggregation over those same properties (and that count is saved on the "count" field of the result). I.e., the results of the new call to `findAll` will be grouped by the properties with data flow, and total counts will be computed for each group. The rest of the rewrite rule is the same as FINDALL-FINDONE, except that the new access in the loop also specifies that the count field should be accessed.

For an example of this transformation, consider the snippets in Figure 4. There is data flow from the video `id` property to the view `videoId` property (line 77), and so the transformed code includes a grouping clause on `videoId` (line 97), and count over `videoId` as well (line 98). To break it down further, the Sequelize line `[Sequelize.fn("COUNT", Sequelize.col("View.videoId")), "count"]` is specifying that a count over `View.videoId` should be issued, and saved under the `count` property of the result. That property is referenced in the loop in the transformed code, on line 101.

FINDALL-FINDBYPK. Calls to `findByPk` take a single argument that is implicitly compared against the primary key of the model being queried. That implicit comparison needs to be made explicit in the new `findAll` query, and so the primary key `pk` of model `M2` is obtained from the model definition. Then, the new call object O_2' can be constructed with a `where` clause that compares the primary key `pk` with a map over the sources `m1s` extracting the relevant field `f` (i.e., the field from the data-flow into the call to `findByPk`). The primary key `pk` is also needed to construct the boolean expression in the find that replaces the old call to `findOne`.

FINDALL-FINDALL. Finally, this rule is nearly identical to the FINDALL-FINDONE rule, the only difference is that instead of performing a `find` over the `m2s` array, a `filter` is performed instead.

Note. The idea that data-flow between ORM API calls is problematic is language-agnostic, and while the rewrite rules use Sequelize API names in them, that is more for readability; the rules represent broader issues in ORMs like finding and then finding again (FINDALL-FINDONE, FINDALL-FINDALL, FINDALL-FINDBYPK), or finding and then counting (FINDALL-COUNT). This is essential functionality to any effective ORM.

$$\frac{\begin{array}{c} props = getAllPropertiesWithDataFlow(\mathtt{O_2}, \mathtt{m1s}) \\ \mathtt{O}'_2 = updatePropReferences(props, \mathtt{O_2}, \mathtt{m1s}, \mathtt{M_1}) \\ \mathtt{BE} = createArrayLookup(props) \qquad \mathtt{m2s}\ fresh \end{array}}{\begin{array}{lcl} \texttt{var m1s = await } \mathtt{M_1}.\underline{\texttt{findAll}}(\mathtt{O_1}) & & \texttt{var m1s = await } \mathtt{M_1}.\underline{\texttt{findAll}}(\mathtt{O_1}) \\ & & \texttt{var m2s = await } \mathtt{M_2}.\underline{\texttt{findAll}}(\mathtt{O}'_2) \\ \texttt{loop \{} & \rightsquigarrow & \texttt{loop \{} \\ \quad \texttt{var m2 = await } \mathtt{M_2}.\underline{\texttt{findOne}}(\mathtt{O_2}) & & \quad \texttt{var m2 = m2s.find(m2 => BE)} \\ \texttt{\}} & & \texttt{\}} \end{array}} \quad \text{(FINDALL-FINDONE)}$$

$$\frac{\begin{array}{c} props = getAllPropertiesWithDataFlow(\mathtt{O_2}, \mathtt{m1s}) \\ \mathtt{O}'_2 = addAggregationAndCount(props, \mathtt{O_2}, \mathtt{m1s}, \mathtt{M_1}) \\ \mathtt{BE} = createArrayLookup(props) \qquad \mathtt{m2s}\ fresh \end{array}}{\begin{array}{lcl} \texttt{var m1s = await } \mathtt{M_1}.\underline{\texttt{findAll}}(\mathtt{O_1}) & & \texttt{var m1s = await } \mathtt{M_1}.\underline{\texttt{findAll}}(\mathtt{O_1}) \\ & & \texttt{var m2s = await } \mathtt{M_2}.\underline{\texttt{findAll}}(\mathtt{O}'_2) \\ \texttt{loop \{} & \rightsquigarrow & \texttt{loop \{} \\ \quad \texttt{var m2 = await } \mathtt{M_2}.\underline{\texttt{count}}(\mathtt{O_2}) & & \quad \texttt{var m2 = m2s.find(m2 => BE).count} \\ \texttt{\}} & & \texttt{\}} \end{array}} \quad \text{(FINDALL-COUNT)}$$

$$\frac{\begin{array}{c} \exists\ dataFlow(m1s, x) \qquad \text{pk primary key of } \mathtt{M2} \\ \mathtt{O}'_2 = \texttt{\{where : \{pk : m1s.map(m1 => m1.f)\}\}} \qquad \mathtt{m2s}\ fresh \end{array}}{\begin{array}{lcl} \texttt{var m1s = await } \mathtt{M_1}.\underline{\texttt{findAll}}(\mathtt{O_1}) & & \texttt{var m1s = await } \mathtt{M_1}.\underline{\texttt{findAll}}(\mathtt{O_1}) \\ & & \texttt{var m2s = await } \mathtt{M_2}.\underline{\texttt{findAll}}(\mathtt{O}'_2) \\ \texttt{loop \{} & \rightsquigarrow & \texttt{loop \{} \\ \quad \texttt{var m2 = await } \mathtt{M_2}.\underline{\texttt{findByPk}}\texttt{(x.f)} & & \quad \texttt{var m2 = m2s.find(m2 => x.f == m2.pk)} \\ \texttt{\}} & & \texttt{\}} \end{array}} \quad \text{(FINDALL-FINDBYPK)}$$

$$\frac{\begin{array}{c} props = getAllPropertiesWithDataFlow(\mathtt{O_2}, \mathtt{m1s}) \\ \mathtt{O}'_2 = updatePropReferences(props, \mathtt{O_2}, \mathtt{m1s}, \mathtt{M_1}) \\ \mathtt{BE} = createArrayLookup(props) \qquad \mathtt{m2s}\ fresh \end{array}}{\begin{array}{lcl} \texttt{var m1s = await } \mathtt{M_1}.\underline{\texttt{findAll}}(\mathtt{O_1}) & & \texttt{var m1s = await } \mathtt{M_1}.\underline{\texttt{findAll}}(\mathtt{O_1}) \\ & & \texttt{var m2s = await } \mathtt{M_2}.\underline{\texttt{findAll}}(\mathtt{O}'_2) \\ \texttt{loop \{} & \rightsquigarrow & \texttt{loop \{} \\ \quad \texttt{var m2 = await } \mathtt{M_2}.\underline{\texttt{findAll}}(\mathtt{O_2}) & & \quad \texttt{var m2 = m2s.filter(m2 => BE)} \\ \texttt{\}} & & \texttt{\}} \end{array}} \quad \text{(FINDALL-FINDALL)}$$

(a) Rewrite rules. ORM API calls are underlined—these calls generate queries. The calls to `find` in the refactored code are essentially maps over the arrays `m2s` that return the element matching the boolean expression specified in the callback.

getAllPropertiesWithDataFlow(O, m) returns all of the properties in an object O that are targets of data-flow from some value m. This will yield a set of property name, value pairs `p : v` for which there exists data-flow between m and the value `v`.

updatePropReferences$(props, O, ms, M)$ creates an object where all of the properties in an object O specified by the list of properties *props* are updated to refer to a map over the array *ms*. I.e., for all property/value pairs `p : v` in *props*, the matching property in O' will be `p : ms.map(m => m.f)`, where `f` is the property of the model M referred to by `v`, either directly in `v` itself (e.g., if `v` is of the form `x.f`) or indirectly in some alias (e.g., if `v = x.f` earlier in the code).

addAggregationAndCount$(props, O, ms, M)$ creates a new object wherein all of the properties in the object O specified by the list of properties *props* are updated to refer to a map over the array *ms*, like *updatePropReferences*. Additionally: (1) a clause is added grouping by all property names p in *props*, and (2) a count aggregation clause is added to total the number of entries in each group.

createArrayLookup$(props)$ builds a boolean expression `BE` to select from the array of results the value that was previously obtained by the query. A property `p : v` has the ORM compare the value of `v` against property `p`, and so a boolean expression `m1.p == v` is created and added with a boolean & to *BE*.

(b) Additional details on helper functions.

Figure 3: Declarative rewrite rule definitions and helper function descriptions. These are discussed in detail in § 3.

4 IMPLEMENTATION

The approach described in § 3 is implemented in a tool called REFORMULATOR. The static data flow analysis is implemented as a taint analysis in CodeQL [24], wherein a taint configuration [25] specifies values returned by ORM API calls as sources, and arguments passed to ORM API calls as sinks. The rewrite rules were implemented using BabelJS [8], a popular JavaScript parser and code generator. Taint flows identified by the analysis are input to the refactoring tool. Sound and scalable static analysis of JavaScript is beyond the current state-of-the-art, and so the code transformations generated by REFORMULATOR are presented to the programmer as suggestions that should be vetted carefully, e.g., by running tests. The code is available in the accompanying artifact [33], which is a Docker image equipped with the ability to re-run the entire evaluation, which is discussed next.

```
64 exports.searchVideo = asyncHandler(async (req, res, next) => {
65   const videos = await Video.findAll ({
66     include: {
67       model: User,
68       attributes : ["id", "avatar", "username"]
69     },
70     where: {
71       title : {
72         [Op.substring ]: req.query.searchterm
73   }}}) ;
74   videos.forEach(async (video, index) => {
75     const views = await View.count({
76       where: {
77         videoId: video.id
78       }
79     }) ;
80     // ...
81   }) ;
82 }) ;
```

```
83 exports.searchVideo = asyncHandler(async (req, res, next) => {
84   const videos = await Video.findAll ({
85     include: {
86       model: User,
87       attributes : ["id", "avatar", "username"]
88     },
89     where: {
90       title : {
91         [Op.substring ]: req.query.searchterm
92   }}}) ;
93   const viewCounts = await View.findAll ({
94     where: {
95       videoId: videos.map(data => data.id)
96     },
97     group: ["View.videoId"],
98     attributes : ["videoId", [Sequelize.fn("COUNT", Sequelize.col("View.videoId")),
         "count"]]
99   }) ;
100  videos.forEach(async (video, index) => {
101    const views = viewCounts.find(x => x.videoId === video.id).count;
102    // ...
103  }) ;
104 }) ;
```

Figure 4: (a) Functionality for search for a video in youtube-clone, where the views for each video are counted in the loop. (b) Refactored version of the code, which generates fewer SQL queries. Note the grouping clause on line 97, and the count attribute on line 98 which sums up the number of elements in each group.

5 EVALUATION

This evaluation of REFORMULATOR aims to answer the following research questions:

(RQ1) How many refactoring opportunities does REFORMULATOR detect?
(RQ2) How often are unwanted behavioral changes introduced by the refactorings suggested by REFORMULATOR?
(RQ3) How do the refactorings affect performance?
(RQ4) How much do the refactorings affect page load times?
(RQ5) What is the running time of REFORMULATOR?

Experimental Setup. We randomly selected 100k JavaScript GitHub repositories that listed Sequelize as an explicit dependency. We then ran the `npm-filter` [7] tool on these repositories to determine how many of them could be automatically installed and built; 37,074 projects satisfied these criteria. We then ran the CodeQL taint analysis on these projects and found 427 projects with N+1 anti-pattern query pairs. From those, we randomly selected projects until we found 8 that we could set up and run with databases populated with meaningful data. Project statistics are listed in Table 1.

Experiment Infrastructure. Experiments were conducted on a 2016 MacBook Pro with 16GB RAM and 2.6 GHz Quad-Core Intel Core i7 processor running MacOS Catalina v10.15.7. The Chrome browser v100.0.4896.127 was used in incognito mode so as to minimize interference from caching and browser extensions.

RQ1: How many refactoring opportunities does REFORMULATOR detect?

To answer this research question, we examined the number of projects in which REFORMULATOR identified anti-patterns. Overall, 427 contained at least one instance of an N+1 anti-pattern from those that built. We examined the distribution of N+1 anti-patterns across the projects; the median number of anti-patterns is 2, and a total of 1,872 anti-pattern instances were detected by the tool. While this is not a huge percentage of the projects (1.1%), the analysis is quite conservative in order to maximize the likelihood of the transformation succeeding.

> REFORMULATOR identified refactoring opportunities in hundreds of GitHub repositories.

RQ2: How often are unwanted behavioral changes introduced by the refactorings suggested by REFORMULATOR?

To answer this research question, we identified which HTTP request handlers in each of the projects contained a refactoring opportunity detected by REFORMULATOR. Every refactoring suggestion was applied to the code. We focused on these handlers as they are the manner in which a front-end would interact with the server; if the handler produces the same response, we deem the behavior to be preserved. There were 44 refactoring opportunities spread across 27 handlers as outlined by columns **# N+1** and **# Handlers** in Table 1. The FINDALL-FINDALL rewrite rule was applied 10 times, FINDALL-FINDBYPK 9 times, FINDALL-FINDONE 5 times, and FINDALL-COUNT 20 times. Note that for this experiment, the databases were populated with test data according to the instructions provided by the repositories.

To conduct the experiment, the UI for each page issuing the HTTP requests and the actual content of the HTTP response was closely examined and compared before and after refactoring. No discrepancies were found, and no refactoring introduced a crash.

> REFORMULATOR did not introduce any unwanted behavioral changes in the applications we studied.

Table 1: Information about subject applications. The first row reads: *the first application is called **youtubeclone**, and commit hash* `47002fc` *was used for the evaluation; **youtubeclone** has 10,551 lines of code spread across 117 files.* REFORMULATOR *detected 12 N+1 pattern query pairs in this application across 7 HTTP request handlers. This is a video-sharing application.*

Project Name	Commit Hash	LOC	# Files	# N+1	# Handlers	Short Description
youtubeclone [23]	`47002fc`	10,551	117	12	7	Video sharing.
eventbright [34]	`e417020`	12,085	122	15	7	Event search and attendance.
property-manage [26]	`33f92a9`	13,959	154	2	2	Property management application.
Math_Fluency_App [29]	`5c1658e`	12,473	114	6	3	Math testing by teachers to students.
employee-tracker [13]	`ba4a195`	10,336	112	3	2	Human resources server API.
Graceshopper-Elektra [14]	`c327530`	12,342	141	1	1	Shopping application.
wall [4]	`ae6c815`	11,152	134	2	2	Image hosting and tagging application.
NetSteam [31]	`5b1cd86`	12,485	136	4	4	Video game trailer viewing application.
			Sum	**44**	**27**	

RQ3: How do the refactorings affect performance?

To answer this research question, we inserted profiling code in the aforementioned HTTP request handlers to collect the time it took the server to prepare a response. We manually interacted with the front-end of each of the subject applications to locate the part of the front-end that sent the request triggering the anti-pattern code. We then restarted the server to empty any server-side caches, triggered the HTTP request again, and collected the time reported by the aforementioned profiling code. We repeated this process ten times before applying the code transformations, and ten more times after: averages and standard deviations of these results are reported in Figure 5 (the error bars represent the average +/- one standard deviation), with each pair of bars corresponding to the time before and after refactoring for a particular HTTP request handler. There are 27 total pairs of bars, corresponding to each of the affected HTTP request handlers, and a link from each "HTTP Request Handler ID" to the code is included in supplemental material.

We found that a low, constant number of queries were issued post-refactoring in all cases, and that every refactoring improved performance. Specifically, we performed a paired two-tailed T-test comparing the 10 run times before and after at 95% confidence and found all differences to be statistically significant. The largest performance gain was in **eventbright**'s handler for getting all events (ID 10, from 279.77ms before to 36.48ms after, an improvement of 7.67x). All HTTP request handlers in **youtubeclone** (IDs 0 through 6) had pronounced improvements, with a median performance improvement of a factor of 2.81x. The smallest benefits were in the **Math_Fluency_App** application (IDs 17 through 19), with a median improvement factor of 1.07x—this is because the number of queries was very small even before refactoring (the number of queries was reduced from 5 to 3, as N was small for this application).

To further understand the performance implications of the refactorings, particularly as database size increased, we conducted a case study involving five request handlers from the 27 in which we refactored instances of the "N+1 problem". In this case study, we created three databases of size 10, 100, and 1000 (henceforth referred to as the "10 scale", "100 scale", and "1000 scale" configurations) so that the HTTP request handler needs to process that much data, and measure the performance of the handlers before and after refactoring at each database size.

The functionality being examined in each application is:

- **youtubeclone**: search for users;
- **eventbright**: main events display;
- **property-manage**: properties dashboard;
- **employee-tracker**: view all employees;
- **NetSteam**: view all reviews for a trailer.

The results of this case study are summarized in Table 3, which reports averages over 10 runs for each database size for each request handler. **youtubeclone**, **eventbright**, and **NetSteam** show dramatic improvements in the relative performance benefits of the refactored code as databases size increases (up to *38.58x* at the 1000 scale for **youtubeclone**). In contrast, the relative performance difference for **property-manage** and **employee-tracker** is not as pronounced with large database sizes; in these applications, most of the time spent serving requests is in processing the data from the database once it is available, rather than waiting for it to become available. Nevertheless, the absolute difference between original and refactored code is substantial at large database sizes even for those two applications, with a 550ms difference for **property-manage** and a nearly 1.5s difference for **employee-tracker**.

All transformations yield statistically significant performance improvements at 95% confidence. Performance gains increase as the size of the database grows; we observed speedups of up to 38.58x.

RQ4: How much do the refactorings affect page load times?

In this research question, we aim to connect the performance improvements observed in serving HTTP requests to measurable improvements in page load time on the client-side.

We conducted a case study on the client-side pages making the HTTP requests studied in the context of **RQ3**. Note: there is no front-end for **employee-tracker**, thus we focus on the other four. The manner in which pages load varies significantly from one application to another, and we found no reliable way to universally time each page load. For example, the **NetSteam** page under study is a pop-up that displays over the main dashboard, and has no URL associated with it, making refresh-based profiling impossible.

Table 2: Information about the run time of REFORMULATOR, with project installation time given for reference. The first row of the table reads: ***youtubeclone*** *took 5.42s to install; it took 24.96s to build the CodeQL database; it took 30.10s to run the N+1 detection query. In total, from a freshly installed* ***youtubeclone***, *REFORMULATOR can run in 55.06s.*

Project Name	Install Time (s)	QLDB Build Time (s)	Query Run Time (s)	Build + Query (s)
youtubeclone [23]	5.42	24.96	30.10	55.06
eventbright [34]	11.42	28.64	32.39	61.03
property-manage [26]	14.68	30.91	33.17	64.08
Math_Fluency_App [29]	4.87	24.41	33.62	58.03
employee-tracker [13]	4.20	23.43	29.41	52.84
Graceshopper-Elektra [14]	24.29	26.69	30.33	57.02
wall [4]	17.29	26.35	29.88	56.23
NetSteam [31]	14.50	29.02	31.79	60.83
Mean	**12.08**	**26.80**	**31.34**	**58.14**

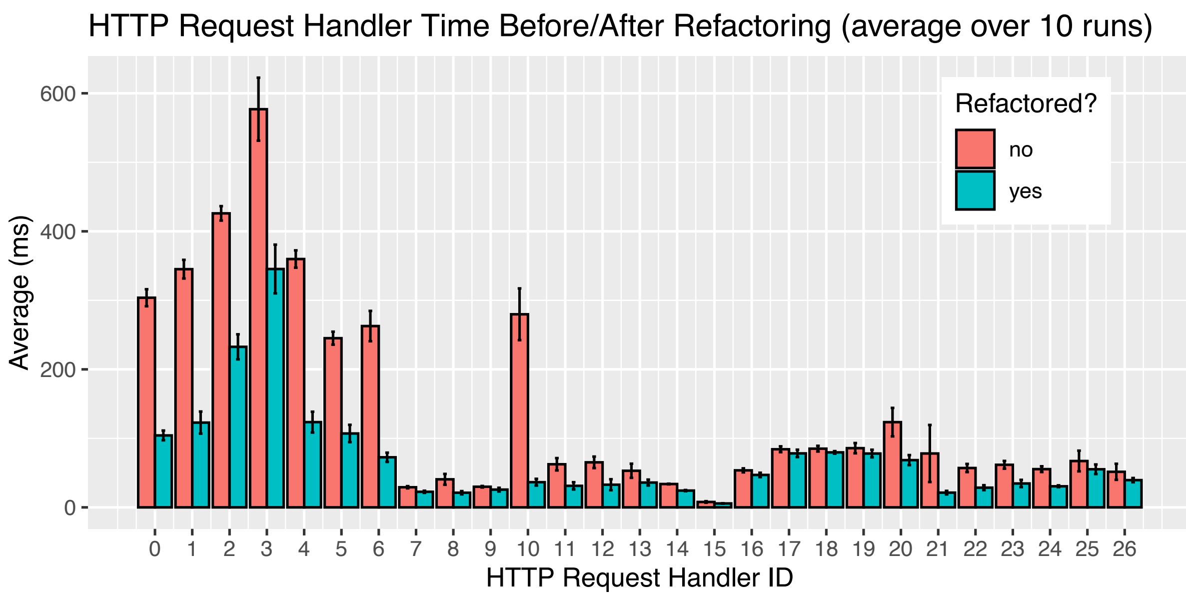

Figure 5: Summary of effect of refactoring on 27 HTTP request handlers. Lower is better. Each pair of bars corresponds to an HTTP request handler. Error bars indicate +/- one standard deviation.

Table 3: Results of case study on 5 applications comparing the scalability of original and refactored code. All times are in ms. The differences were all statistically significant (paired two-tailed T-test at 95% confidence); standard deviations are omitted for brevity, and can be found in supplemental material. The first row of the table reads: *for test ID 1 in the* ***youtubeclone*** *application, with a database size of 10, the mean before refactoring is 360.30ms, and after refactoring is 118.06ms; this represents a performance improvement with a factor of 3.05x (= 360.30 ÷ 118.06).*

		DB Size = 10			***DB Size = 100***			***DB Size = 1000***		
Project Name	**ID**	**Before**	**After**	***Scale***	**Before**	**After**	***Scale***	**Before**	**After**	***Scale***
youtubeclone [23]	1	360.30	118.06	*3.05x*	1937.42	152.96	*12.67x*	18171.86	471.07	*38.58x*
eventbright [34]	10	111.38	31.94	*3.49x*	797.35	49.53	*16.10x*	7001.48	214.61	*32.62x*
property-manage [26]	20	56.91	33.71	*1.69x*	246.06	111.05	*2.22x*	1333.64	786.44	*1.70x*
employee-tracker [13]	14	57.15	34.32	*1.67x*	374.73	153.97	*2.43x*	2495.92	1010.47	*2.47x*
NetSteam [31]	21	77.05	39.01	*1.98x*	337.67	41.62	*8.11x*	2129.34	108.06	*19.71x*

Further, most web profiling tools rely on the collection of a trace as a page loads, and that trace includes a variable number of frames *before* the page begins to refresh, leading to unfortunate variability and inaccuracies in performance numbers collected automatically.

In light of this, we opted to manually study the behavior of each page with Chrome DevTools [16] to obtain rich information about how each page behaves, paying particular attention to the "Performance" and "Network" tabs. The times reported are estimations based on the trace timeline displayed by the "Performance" tab of the Chrome DevTools (label (E) in Figure 6) that displays a timeline of screenshots of a page, which we believe corresponds most closely with the observable user experience. Specifically, as in our study of **RQ3**, we triggered each HTTP request 10 times and estimated the time between when the request was triggered and when the page was visibly populated with data; we drew these estimates from the time markers in the timeline, and rounded to the nearest quarter second, and averages are reported throughout this section. We examined the behavior of each page at three database scales (10, 100, and 1000), and report on our findings below. Screenshots of the DevTools profiles used in this study as well as raw observations are included in supplemental material.

youtubeclone *(search for users).* In this application, we found the network time to be the limiting factor in the client page being fully rendered, as the page was quickly populated once all the data was returned from the server. At the 1000 scale, the difference in load time was dramatic (19.88s with the original code vs. 1.9s with the refactored code, a ~10x improvement). The difference in load time is also very noticeable at the 100 scale, and a screenshot comparing the effect of the transformation on the load time can be found in Figure 6 (3.8s before refactoring vs. 0.8s with refactoring). Even at the smaller 10 scale, appreciable load time improvements were observed (from 1.2s to 0.5s).

eventbright *(main events display).* The front-end is quickly populated with data once it is received from the server. We noted dramatic load time improvements at the 1000 scale (7.7s with original code vs. 1.4s with refactored code), and a noticeable improvement at the 100 scale (1s with original code vs. 0.3s with refactored code), and a very small difference at the 10 scale (0.4s before vs. 0.3s after).

property-manage *(property dashboard).* In this application, the refactoring did not appear to affect the load time of the page. Even at the 1000 scale, the dashboard took nearly 3s to be populated with data, even though the server finished fully processing the request 1.5s faster in the refactored version. This is because the information computed by the ORM API call in the loop is used internally by the server, and is not part of the response.

In spite of this, the refactoring is still beneficial: as applications move away from locally-hosted databases, the number of concurrent database requests becomes a concern, as many remote database management systems only allow up to a certain number of requests simultaneously, after which point requests are refused. The refactoring proposed by REFORMULATOR reduces the number of requests here from N+1 (with N being the number of properties) to two.

NetSteam *(reviews for trailer).* Here, a dashboard presents many video game titles to the user, and the user may select one of them to bring up an animated pop-up with the trailer and reviews for the game. At the 1000 scale, it took 3.8s on average for the reviews to load with the original code vs. 2s with the refactored code. At the 100 and 10 scales, the animation displaying the trailer and reviews masked any performance difference between original and refactored code, as the animation completes before the reviews load at both scales before and after refactoring.

In several cases, the refactoring suggested by REFORMULATOR results in dramatic speedups (of up to 90%).

RQ5: What is the running time of REFORMULATOR?

Table 2 shows the time it takes `npm install` to install the project's dependencies (given for reference, column **Install Time**), the time it takes to build the CodeQL database, which is needed to run any CodeQL queries on the code (**QLDB Build Time**), and the time to run REFORMULATOR's anti-pattern detection query (**Query Run Time**). The time taken to build the QLDBs *and also* run the queries is consistently between 50 and 65 seconds. The time to run the actual code transformation is less less than a second in all cases and is not reported in the table.

The running time of REFORMULATOR on a fresh installation of a project is 58.14s on average.

6 THREATS TO VALIDITY

We have identified some threats to the validity of our work.

The primary threat to validity is the fact that the transformations proposed by our tool may not preserve program behavior. Static analysis of JavaScript is unsound due to the extreme dynamicity of the language, as rampant dynamic property redefinition, event-driven programming, and promise-based asynchrony have made precise and scalable analysis elusive. REFORMULATOR is a tool that leverages static program analysis, and is thus unsound; we have accepted this in designing REFORMULATOR, and focused on developing a tool that is practical. During the course of our evaluation, we found that no behavior-altering transformations were suggested.

It is also possible that our selection of projects for evaluation is not representative. We mitigate this by selecting projects randomly from those that explicitly declare Sequelize as a dependency. This list was pruned to find projects that could be successfully built and for which we could configure and populate databases, but this was entirely so that the effect of the transformations could be studied.

7 RELATED WORK

There is a large body of existing research aimed at improving the performance of database-backed applications, including database refactoring, bug detection, and query optimization.

Database refactoring. Existing work has considered refactoring database schemas to improve performance. Ambler and Sadalage [5] catalogue *database refactorings*, i.e., behavior-preserving changes to a database schema such as moving a column from one table to another. Similarly, Xie et al. [36] and Wang et al. [35] study how application code must be updated in response to schema changes. Rahmani et al. [28] present an approach for avoiding serializability violations in database applications by transforming a program's

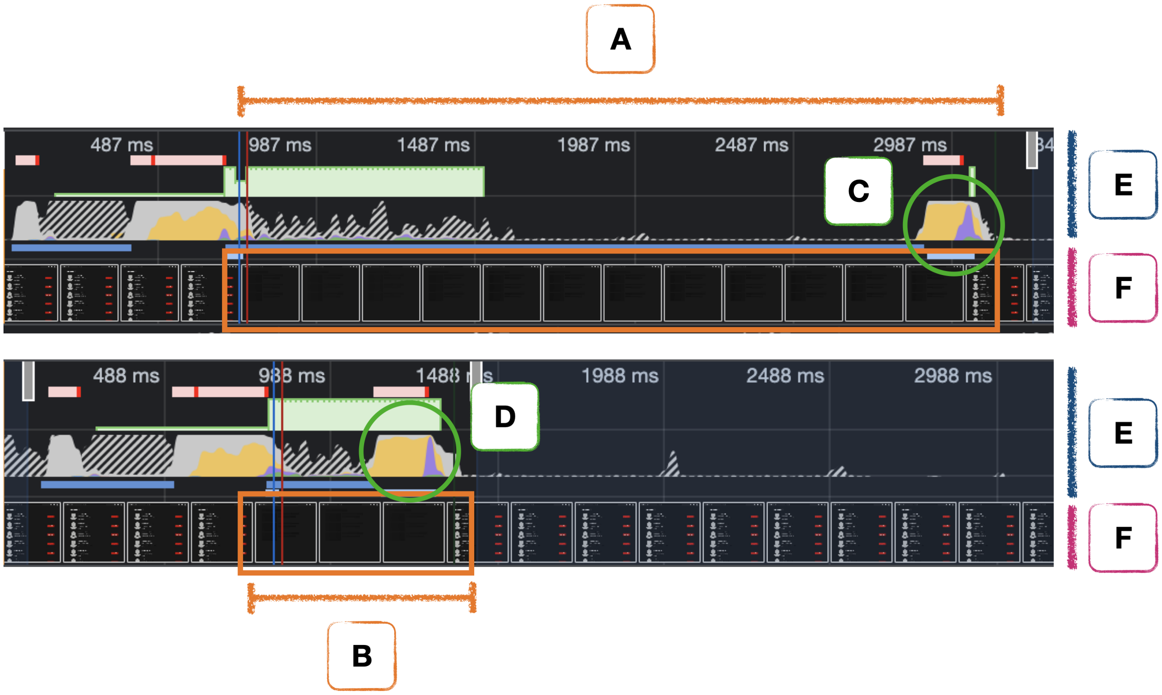

Figure 6: Two screenshots from the Chrome DevTools' Performance Tab profiling a search turning up 100 users in youtubeclone. The profile corresponding to the original code is on top, and the refactored one is on the bottom. The two (E) labels show time series of application activity, where higher values correspond to more CPU cycles. (C) and (D) show spikes in activity when the HTTP response was received by the client before and after refactoring, resp. The two (F) labels show a series of screenshots taken of the front-end as it loads and is populated by data. (A) and (B) show the period that the screen was idle before and after refactoring, resp., and the two boxes in the timelines highlight that the screen is empty during that span.

data layout. This nature of work provides insight into the relationship between database structure and performance, but does not consider query-based performance bugs like the 'N+1 problem'.

Identifying the 'N+1 problem' in database code. Yang et al. [38] use dynamic analysis to detect performance anti-patterns in Ruby on Rails [30] applications and manually refactor them to assess performance impact. One of these anti-patterns, "inefficient lazy loading", is a variant of the 'N+1 problem' they report to be prevalent in their experiments. Chen et al. [9] report on industrial experience, observing 17 ORM-related performance problems in PHP applications that use the Laravel ORM [2], including the same "inefficient lazy loading" anti-pattern. Chen et al. [10] use static analysis to detect anti-patterns in JPA, a popular ORM for Java, including "one-by-one processing" where a list of objects of one class is iterated over, and objects from another class are found by issuing a SELECT query. Their proposed resolution involves introducing *batching* (i.e., waiting for several queries to be created before issuing them all at once). Cheung et al. [12] created a "lazy-ifying" compiler that also batches queries to reduce the number of round trips to the database. Batching queries does alleviate the 'N+1 problem' by reducing the amount of database round-trips, but it does not eliminate the problem through permanent refactoring.

Identifying other performance bugs in database code. Chen et al. [11] consider situations where calling the API of the Hibernate ORM [18] for Java results in accessing redundant data (e.g., some columns in a table need to be updated, but a query is generated that updates *all* of them). They assess performance impact by manually rewriting subject applications. Yan et al. [37] identify optimization opportunities in Ruby on Rails [30] applications using static analysis and profiling, including a "Fusing queries" optimization targeting situations where the result of a query flows into another query. Yang et al. [40] present a framework in which static analysis and dynamic profiling are used to visualize, for each HTML tag, the set of database queries needed to generate the data needed to render it. Their framework also suggests view-changing refactorings (e.g., introducing pagination) to improve performance. While there is much work on detecting query-based performance bugs, including the 'N+1 problem', using static and dynamic analysis, this work leaves actual optimization to manual refactoring.

We know of 2 research efforts to use static analysis to automatically refactor source code to remove database bugs. Yang et al. [39] design a RubyMine IDE plugin named PowerStation which uses static analysis to identify and refactor common ORM performance inefficiencies. While this work relates most closely to ours, PowerStation does not identify or refactor the 'N+1 problem'. Instead, PowerStation tackles other inefficiencies like dead stores, redundant loads, and Ruby-specific API misuses. Lyu et al. [22] present an automatic refactoring technique for repetitive autocommit transactions, using static analysis to detect this database ineffiency common to the Android platform. However, repetitive autocommit transactions refer to writes, whereas the 'N+1 problem' concerns reads.

In sum, previous work explored database-related refactorings and the detection of ORM anti-patterns. However, we are not aware of automated refactoring tools for eliminating the 'N+1 problem'.

8 CONCLUSION

ORMs provide an object-oriented interface to databases and facilitate the development of database-backed applications. In an ORM, databases can be accessed using method calls to the ORM, which maps those calls into database queries. While convenient, this added layer of abstraction hides the significant performance cost of database operations, and misuse of ORMs can lead to far more queries being generated than necessary. In particular, the "N+1 problem" is prevalent in ORM-backed applications. It is natural to iterate over collections in object-oriented languages, but iterating over data that originates from a database and calling an ORM method in each iteration may result in suboptimal performance. In such cases, it is often possible to reduce the number of round-trips to the database by issuing a single query that fetches all desired results at once.

In this work, we presented an approach for automatically refactoring applications that use ORMs to eliminate instances of the "N+1 problem", which relies on static analysis to detect data flow between ORM API calls. We implemented this approach in REFORMULATOR, a tool targeting the Sequelize ORM in JavaScript, and evaluated it on 8 JavaScript projects. We found 44 N+1 query pairs in these projects, and REFORMULATOR refactored all of them successfully, resulting in improved performance while preserving program behavior. At a small scale, performance improvements of up to 7.67x were observed, and improvements of up to 38.58x were observed at scale. Further, a detailed study of the front-ends of these applications revealed page load time improvements of up to 90%.

ACKNOWLEDGMENTS

A. Turcotte was supported in part by NSERC. A. Turcotte and F. Tip were supported by NSF grant CCF-1907727. Authors would like to thank Oracle Labs for their support.

REFERENCES

[1] 2021. ECMAScript 2021 Language Specification Section 27.2: Promises. https://262.ecma-international.org/#sec-promise-objects.

[2] 2022. Laravel: The PHP Framework for Web Artisans. See https://laravel.com/.

[3] 2022. Sequelize ORM. See https://sequelize.org.

[4] adam dill. 2022. wall. See https://github.com/adam-dill/wall/commit/ae6c815.

[5] Scott Ambler and Pramod Sadalage. 2006. *Refactoring Databases: Evolutionary Database Design* (1 ed.). Addison-Wesley.

[6] Ellen Arteca, Frank Tip, and Max Schäfer. 2021. Enabling Additional Parallelism in Asynchronous JavaScript Applications. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:28. https://doi.org/10.4230/LIPIcs.ECOOP.2021.7

[7] Ellen Arteca and Alexi Turcotte. 2022. npm-filter: Automating the mining of dynamic information from npm packages. *arXiv preprint arXiv:2201.08452* (2022).

[8] Babel. 2022. Babel. See https://babeljs.io/.

[9] Boyuan Chen, Zhen Ming Jiang, Paul Matos, and Michael Lacaria. 2019. An Industrial Experience Report on Performance-Aware Refactoring on a Database-Centric Web Application. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 653–664. https://doi.org/10.1109/ASE.2019.00066

[10] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting Performance Anti-Patterns for Applications Developed Using Object-Relational Mapping. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 1001–1012. https://doi.org/10.1145/2568225.2568259

[11] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2016. Finding and Evaluating the Performance Impact of Redundant Data Access for Applications that are Developed Using Object-Relational Mapping Frameworks. *IEEE Transactions on Software Engineering* 42, 12 (2016), 1148–1161. https://doi.org/10.1109/TSE.2016.2553039

[12] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. 2014. SLOTH: Being Lazy Is a Virtue (When Issuing Database Queries). In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 931–942. https://doi.org/10.1145/2588555.2593672

[13] daedadev. 2022. employee-tracker. See https://github.com/daedadev/employee-tracker/commit/ba4a195.

[14] Elektra-GHP. 2022. Graceshopper-Elektra. See https://github.com/Elektra-GHP/Graceshopper-Elektra/commit/c327530.

[15] Satyajit Gokhale, Alexi Turcotte, and Frank Tip. 2021. Automatic migration from synchronous to asynchronous JavaScript APIs. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. https://doi.org/10.1145/3485537

[16] Google. 2022. Chrome DevTools. See https://developer.chrome.com/docs/devtools/.

[17] Neville Grech and Yannis Smaragdakis. 2017. P/taint: Unified points-to and taint analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.

[18] Hibernate. 2022. What is Object/Relational Mapping? See http://hibernate.org/orm/what-is-an-orm/.

[19] Rezwana Karim, Frank Tip, Alena Sochurková, and Koushik Sen. 2020. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Trans. Software Eng.* 46, 12 (2020), 1364–1379. https://doi.org/10.1109/TSE.2018.2878020

[20] Hee Yeon Kim, Ji Hoon Kim, Ho Kyun Oh, Beom Jin Lee, Si Woo Mun, Jeong Hoon Shin, and Kyounggon Kim. 2022. DAPP: automatic detection and analysis of prototype pollution vulnerability in Node.js modules. *Int. J. Inf. Sec.* 21, 1 (2022), 1–23. https://doi.org/10.1007/s10207-020-00537-0

[21] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2021. Detecting Node.js Prototype Pollution Vulnerabilities via Object Lookup Analysis. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 268–279. https://doi.org/10.1145/3468264.3468542

[22] Yingjun Lyu, Ding Li, and William G. J. Halfond. 2018. Remove RATs from Your Code: Automated Optimization of Resource Inefficient Database Writes for Mobile Applications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 310–321. https://doi.org/10.1145/3213846.3213865

[23] manikandanraji. 2022. youtubeclone. See https://github.com/manikandanraji/youtubeclone-backend/commit/47002fc.

[24] Microsoft. 2022. CodeQL. See https://codeql.github.com/.

[25] Microsoft. 2022. CodeQL. See https://codeql.github.com/docs/codeql-language-guides/analyzing-data-flow-in-javascript-and-typescript/#analyzing-data-flow-in-javascript-and-typescript.

[26] mikethecodegeek. 2022. property-manage. See https://github.com/mikethecodegeek/property-manage/commit/33f92a9.

[27] Joonyoung Park, Inho Lim, and Sukyoung Ryu. 2016. Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 61–70. https://doi.org/10.1145/2889160.2889227

[28] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. 2021. Repairing Serializability Bugs in Distributed Database Programs via Automated Schema Refactoring. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 32–47. https://doi.org/10.1145/3453483.3454028

[29] rayace5. 2022. Math_Fluency_App. See https://github.com/rayace5/Math_Fluency_App/commit/5c1658e.

[30] Ruby on Rails. 2022. Ruby on Rails. See https://rubyonrails.org/.

[31] W the V. 2022. NetSteam. See https://github.com/W-the-V/NetSteam/commit/5b1cd86.

[32] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. *ACM Sigplan Notices* 44, 6 (2009), 87–97.

[33] Alexi Turcotte, Mark W. Aldrich, and Frank Tip. 2022. *Reformulator: Artifact.* https://doi.org/10.5281/zenodo.6959485

[34] twincarlos. 2022. eventbright. See https://github.com/twincarlos/eventbright/commit/e417020.

[35] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing Database Programs for Schema Refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 286–300. https://doi.org/10.1145/3314221.3314588

[36] Sophie Xie, Junwen Yang, and Shan Lu. 2021. Automated Code Refactoring upon Database-Schema Changes in Web Applications. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 1262–1265. https://doi.org/10.1109/ASE51524.2021.9678934

[37] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2017. Understanding Database Performance Inefficiencies in Real-World Web Applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management* (Singapore, Singapore) *(CIKM '17)*. Association for Computing Machinery, New York, NY, USA, 1299–1308. https://doi.org/10.1145/3132847.3132954

[38] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How Not to Structure Your Database-Backed Web Applications: A Study of Performance Bugs in the Wild. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 800–810. https://doi.org/10.1145/3180155.3180194

[39] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. 2018. 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*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 884–887. https://doi.org/10.1145/3236024.3264589

[40] Junwen Yang, Cong Yan, Chengcheng Wan, Shan Lu, and Alvin Cheung. 2019. View-Centric Performance Optimization for Database-Backed Web Applications. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 994–1004. https://doi.org/10.1109/ICSE.2019.00104