Automated Code Refactoring upon Database-Schema Changes in Web Applications

Sophie Xie*, Junwen Yang†, Shan Lu†
*Whitney Young High School, sxie2@cps.edu, †University of Chicago, {junwen, shanlu}@uchicago.edu

Abstract—Modern web applications manipulate a large amount of user data and undergo frequent data-schema changes. These changes bring up a unique refactoring task: updating application code to be consistent with data schema. Previous study and our own investigation show that this type of refactoring is error-prone and time-consuming for developers. This paper presents EvolutionSaver, a static code analysis and transformation tool that automates schema-related code refactoring and consistency checking. EvolutionSaver is implemented as an IDE plugin that works for both Rails and Django applications. The source code of EvolutionSaver is available on Github [1] and the plugin can be downloaded from Visual Studio Marketplace [2], with its tutorial available at https://www.youtube.com/watch?v=qBiMKLFjbeE and DOI 10.5281/zenodo.5276127.

I. INTRODUCTION

Modern web applications often use database engines to manage a large amount of user data, such as user profiles in social network applications and transaction records in on-line shopping platforms [3]. The schema of such data goes through changes, such as table renaming, column deletion, and others, for better performance or functionality when an application evolves [4]. Unfortunately, it is difficult for developers to keep their code consistent with database schema changes all the time, a task we refer to as schema-related code refactoring, with any inconsistency leading to application crashes.

Schema-related refactoring and traditional refactoring like class renaming share similarities, given that popular Object Relational Mapping (ORM) frameworks, such as Rails [5], Django [6], and Hibernate [7], allow database data to be updated and retrieved in an object-oriented way—the name of a database table corresponds to the name of a model class and the names of table columns are the same as class fields.

However, they also differ in various aspects, due to the unique nature of persistent data, as we elaborate below.

How is schema defined? Different from a regular class whose field names and field types are defined by its class declaration, a model class’s structure has to match its corresponding database table that is created once at an application’s installation or upgrade. In fact, in some ORM frameworks like Rails, persistent fields of a model class are not declared in its class definition and are instead automatically mapped by Rails from the corresponding table schema, which is defined through ORM APIs like create_table in Rails or CreateModel in Django in a type of files called migration files, as shown in Listing [1].

Listing 1. Migration files from Onebody

```ruby
] Creating a new schema
1 create_table :people do |t|
2 t.integer :id
3 t.text :sequence
4 rename_column :people, :sequence, :position
```

Listing 2. Inconsistent code from Onebody

```ruby
] Renaming a column
1 @family.people.undeleted.order('sequence')
2 person.sequence = 1
```

How is schema changed? Schema changes are expressed through ORM APIs in migration files (e.g., line 4 in Listing [1] renames the column sequence in table people), which informs the web application about how to update its database during installation and upgrade. In an ORM framework like Rails, schema changes cannot be seen in class definitions.

What code refactoring is needed? Following a schema change in the migration file, corresponding references in the application need to change. Some of these are just class or field renaming like in line 2 of Listing [2] while some require changing ORM APIs’ parameters like in line 1 of Listing [2].

For example, developers of Onebody [8], a popular social network application, used a table people to keep user information. In one commit, they renamed the sequence column in people to position (line 4 in Listing [1]). In the same commit, they correctly updated the reference to sequence in 6 places across 4 files like the one in line 2 of Listing [2] but forgot to change the other 5 places, such as the parameter reference.

Recent work motivated tool support for schema-related refactoring [4] and proposed techniques to synthesize updates to a list of SQL queries given the old schema and the new schema written in SQL [9]. Although inspiring, it does not directly help many web applications, whose schema changes and database operations are expressed in ORM APIs, rarely if ever in raw SQL.

This paper presents EvolutionSaver, a tool that uses ORM-aware static analysis to help schema-related code refactoring in web applications written in Rails [5] and Django [6], two popular web frameworks. Given two versions of a web application, EvolutionSaver analyzes and identifies schema changes from migration files, searches for any code inconsistent with the
new schema, and generates warnings and patches accordingly.

To ease its adoption, we have integrated EvolutionSaver into the popular Visual Studio Code IDE [10] as a plugin. Web developers can use this plugin to guide their schema-related refactoring or to look for schema–code inconsistency bugs.

In our evaluation with 12 popular Rails and Django applications, EvolutionSaver detected 86 schema–code inconsistencies caused by 35 schema changes in the past. We have reported 11 of them that exist in the latest versions to developers, and got 10 of them already confirmed and 6 of them already patched based on our suggestion. Our examination of the rest 75 inconsistencies shows that they took many days for developers to discover and fix.

EvolutionSaver’s source code is on Github [11] and the plugin can be downloaded from Visual Studio Marketplace [2].

II. BACKGROUND AND EXTENDED MOTIVATION

Background. A web application’s schema gets initialized and updated by migration APIs in migration files, a mechanism supported by ORM frameworks. For example, Listing 1 illustrates two migration files, each with one migration API call: the first creates a table named people with two columns id and sequence, which are automatically mapped to two fields in a corresponding model class with a singular name (Person class); the second renames a table column, which automatically causes a field name change in its model class.

During an installation/upgrade of a web application, the ORM framework executes all the latest migration files not yet executed on this installation, calling migration APIs in these files one by one and updating the schema along the way.

Extended motivation. A recent study [4] on 100 Rails applications showed that most applications went through many schema changes. For further motivation, we studied 12 Rails and Django web applications from different categories like forum, e-commerce, social network, etc. [2] They are all highly rated, each with more than 1000 stars on GitHub, 11,000–900,000 lines of code and 500–100,000 commits.

How often are schema changes? 18% – 85% of application versions contain at least one schema change, and there are more than 8 changes for every version on average. Furthermore, changes are common throughout the development history of each application. For example, across the 6 Rails applications, the most recent 25% of commits happen to contain about 25% of all the schema changes in total.

What types of changes are there? As shown in Table I, changes to various aspects of the schema are all common. About three quarters of changes add tables, columns, or indices, and do not directly cause inconsistency with existing code. The remaining one quarter of changes modify or delete existing tables, columns, associations, or indices, immediately threatening code consistency, and hence are the target of EvolutionSaver, as detailed in Table II.

A. Schema change extraction

Different from previous work [9] that identifies schema changes by comparing the old and the new schemas written in SQL, EvolutionSaver takes the unique opportunity offered by web applications and identifies schema changes directly from all the new migration files in the new version.

Specifically, 12 out of 19 Rails migration APIs and 6 out of 17 Django migration APIs introduce schema changes that can immediately cause code inconsistency. EvolutionSaver matches each such API, or an API–parameter combination in case of Django, with one change type listed in Table II. Whenever such an API call is identified in a migration file, EvolutionSaver extracts related change information, like table and column names, and saves it as a change record for later use. EvolutionSaver aggregates related changes to the same target: deleting a column and then adding it back to the same table will be aggregated and correctly considered as no change.

Finally, in Rails, since an association relationship is defined partly through model classes, EvolutionSaver compares model class definitions, in addition to migration files, to get association changes. For example, the model class definitions in Listing 3 uses has_many to indicate that each User record is related to multiple Comment records, which can be retrieved through the association field comments defined in User.

B. Query extraction

Next, EvolutionSaver identifies all the queries that can be issued by the new version of the application. In ORM, a query can be expressed in two forms: 1) an ORM query API such as find_by in Rails and filter in Django invoked upon either an object holding a previous query’s result or a model class like that in line 1 of Listing 3, 2) the reference to a model class’ association field. For example, following the association definition in Listing 3, the query select * from comments where user_id = user.id from comments where user_id = user.id.

EvolutionSaver identifies both forms of queries and extracts the names of table, column, index, and related association information from each query. The analysis for Django applications is done by analyzing the AST generated by pyast. The analysis for Rails applications is built upon ORM-aware static analysis framework PowerStation [13]. EvolutionSaver uses

<table>
<thead>
<tr>
<th>Change Target</th>
<th>Table</th>
<th>Column</th>
<th>Association</th>
<th>Index</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Django</td>
<td>1.0</td>
<td>4.4</td>
<td>2.6</td>
<td>0.2</td>
<td>8.2</td>
</tr>
<tr>
<td>Rails</td>
<td>1.6</td>
<td>4.8</td>
<td>2.2</td>
<td>1.3</td>
<td>9.9</td>
</tr>
</tbody>
</table>

TABLE I
NUMBER OF SCHEMA CHANGES PER VERSION
Listing 3. Association-Field Definition and Changes in Onebody

```ruby
class User
  has_many :comments, :class_name => 'Comment'
  + has_many :real_comments, :class_name => 'Comment'
end

class Comment
  belong_to :user, :class_name => 'User'
end
```

intra-procedural dependency analysis to identify objects that hold results from a previous query. In theory, it may miss queries on objects defined in a different procedure.

C. Inconsistency detection and refactoring suggestion

Finally, EvolutionSaver goes through each schema change, searches for inconsistency with queries in the new version, and generates refactoring suggestion accordingly (Table II).

**Table II**

<table>
<thead>
<tr>
<th>Changes</th>
<th>Change Target</th>
<th>EvolutionSaver Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name Change</td>
<td>Column</td>
<td>Refactoring suggestion</td>
</tr>
<tr>
<td>Association</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deletion</td>
<td>Column</td>
<td>Refactoring suggestion</td>
</tr>
<tr>
<td></td>
<td>Association</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>Inconsistency warning</td>
</tr>
<tr>
<td>Type Change</td>
<td>Column</td>
<td>Inconsistency warning</td>
</tr>
<tr>
<td></td>
<td>Association</td>
<td>Partial refactoring suggestion</td>
</tr>
</tbody>
</table>

**A. Features of EvolutionSaver IDE plugin**

We have implemented EvolutionSaver as a plugin for Visual Studio Code [4], a popular IDE for multiple languages. As shown in Figure 1, one can press the start button to start the plugin. By default, EvolutionSaver compares the current code with the latest commit. Users can also specify the commits they desire to check in the configuration file.

**Issue list.** The left panel, as shown in Figure 1, lists all the errors detected by EvolutionSaver in a hierarchical view. The first level lists the files where errors are detected; clicking a file shows the details of every error in that file, including the line number and the type of root-cause schema change; clicking the error shows a “Fix” button.

**Issue detail.** Clicking a file in the issue list will navigate users to the corresponding file in the editor, with the error code highlighted. Users can hover their mouses over the highlighted code and see the detailed explanation, like “Group-Time.ordering is RENAMED TO sequence” in Figure 1.

**Issues fix.** One can click the ‘Fix’ button on the left panel or a ‘Fix all’ button to fix one or all the issues.

**B. Implementation**

The start button triggers our static analyzer to run on the given code commits. The analyzer produces an output.json file that is parsed to create the issue list using the Visual Studio Code Extension APIs TreeDataProvider and TreeItem. The DocumentHighlightProvider API is used to highlight the selected error code, given the filename and line number information in output.json. The HoverProvider API enables the tooltip of detailed reason to display once hovering over the highlighted code. To fix the error, TextDocument, Range, and ExtensionContext APIs are used to insert, replace, and delete source code in the editor panel.

**V. EVALUATION AND THREATS TO VALIDITY**

We have evaluated EvolutionSaver using 6 Ruby-on-Rails applications and 6 Django applications (the same 12 applications described in Section I). For each application, we apply EvolutionSaver on all its consecutive commits.

**Detection.** As shown in Table III, EvolutionSaver automatically identifies 86 inconsistency errors caused by 35 schema changes.
changes, with no false positives based on our manual examination. Among them, 11 errors exist in the latest versions. After reporting them to developers, 10 already got confirmed. These 11 errors have existed for 234 days on average (median: 61 days) when reported by us. The other 75 errors on average existed in these applications for 232 days (median: 7 days) and 409 commits (median: 16 commits) until finally discovered and fixed by developers. In theory, some developers may intentionally split schema changes and follow-up code changes to separate commits. This is unlikely for most of these errors given the long gap taken to fix them. Moreover, about half of them were not fixed until after major code releases.

Refactoring. Among the 11 errors that we reported to developers, for 6 of them, developers already accepted the refactoring patches suggested by EvolutionSaver and merged them into the main branch. For the other 75 errors that were fixed by developers in the past, the related statements, functions, or files were often deleted in the fixed commit. There are 13 of them where related code regions still existed in the fixed commit, and these 13 fixes are exactly the same as the refactoring suggested by EvolutionSaver.

Performance. EvolutionSaver takes 3–125 seconds (35s on average) to process consecutive commits of Rails applications with 11,000–900,000 lines of code, and 1–40 seconds (19s on average) for Django apps with 17,000–174,000 lines of code.

Threats to validity. As discussed in Section III, EvolutionSaver may raise false alarms in rare cases. There are also sources of false negatives. Application code that cannot be parsed by pyast or Yard cannot be analyzed by EvolutionSaver. A schema may be changed by SQL commands issued directly to the database without any record in migration files. This is considered a bad practice [15] and is not handled by EvolutionSaver. A schema may also be changed in migration files through raw SQL commands wrapped in ORM APIs like migrations.RunSQL(...) in Django and migrations.execute(...) in Rails. This feature is rarely used by web developers (less than 1% of cases in our study), and is not handled by EvolutionSaver. If the new version adds a table T, and then changes the schema about T or its columns, indices, or associations, EvolutionSaver would not check whether the new code is consistent with the schema of T, as T does not exist in the old version. Finally, what we observed in the 12 Rails and Django applications may not apply to other open-source applications.

VI. RELATED WORK
Recent work uses ORM-aware static analysis to detect performance anti-patterns [13, 16] and data constraint problems [17] in database-backed web applications. They did not look at schema changes and are orthogonal to our work. EvolutionSaver is motivated by recent work [4, 9] about schema changes in web applications, but is different from them as discussed earlier. Specifically, MIGRATOR [9] analyzes schema changes in SQL and synthesizes SQL queries, while EvolutionSaver looks at Rails (Ruby) and Django (Python) application; MIGRATOR handles renaming changes and structure changes like moving a column from one table to another, while EvolutionSaver handles all the changes in Table II.

VII. CONCLUSION
EvolutionSaver is a static analysis tool that detects schema-code inconsistency and suggests refactoring in web applications. Evaluation shows that EvolutionSaver is effective in analyzing large open-source Rails and Django applications.

VIII. ACKNOWLEDGEMENTS
This work is supported in part by the National Science Foundation through grants CNS-1956180, CCF-1837120, CNS-1764039, CNS-1563956, and IIS-1546543; the Marian and Stuart Rice Research Award; gifts from Facebook Research, and the CERES Center for Unstoppable Computing.


REFERENCES