SLocator: Localizing the Origin of SQL Queries in Database-Backed Web Applications

Wei Liu, Tse-Hsun (Peter) Chen

Abstract—In database-backed web applications, developers often leverage Object-Relational Mapping (ORM) frameworks for database accesses. ORM frameworks provide an abstraction of the underlying database access details so that developers can focus on implementing the business logic of the application. However, due to the abstraction, developers may not know where and how a problematic SQL query is generated in the application code, causing challenges in debugging database access problems. In this paper, we propose an approach, called SLocator, which locates where a SQL query is generated in the application code. SLocator is a hybrid approach that leverages both static analysis and information retrieval (IR) techniques. SLocator uses static analysis to infer the database access for every possible path in the control flow graph. Then, given a SQL query, SLocator applies IR techniques to find the control flow path (i.e., a sequence of methods called in an interprocedural control flow graph) whose inferred database access has the highest similarity ranking. We implement SLocator for Java's official ORM API specification (JPA) and evaluate SLocator on seven open source Java applications. We find that SLocator is able to locate the control flow path that generates a SQL query with a Top@1 accuracy ranging from 37.4% to 70% for SQL queries in sessions, and 30.7% to 69.2% for individual SQL queries; and Top@5 ranging from 78.3% to 95.5% for SQL queries in sessions, and 59.1% to 100% for individual SQL queries. We also conduct a study to illustrate how SLocator may be used for locating issues in the database access code.

Index Terms—Localization, Static Analysis, Information Retrieval, Object-Relational Mapping

1 Introduction

Modern database-backed web applications are becoming more complex due to the ever-increasing functionality. To reduce development efforts and allow developers to focus on the business logic of the applications, database-backed web applications often use Object-Relational Mapping (ORM) frameworks to abstract database accesses. ORM frameworks have become increasingly popular with implementations in most modern programming languages such as Java, C#, Python, and Ruby [1], [2]. A report also shows that among the 2,164 surveyed Java developers, ORMs are the leading means of database access and 67.5% use Hibernate (one of the most popular Java ORM frameworks) instead of other database abstraction frameworks [3]. ORM provides a conceptual mapping between objects in object-oriented programming languages, such as Java, and tables in database management systems (DBMSs). With ORM mapping, developers can access the DBMS through a combination of object modifications and ORM API calls. For example, by calling user.setName("Alice") followed by entityManager.persist(user), the ORM framework would automatically generate a SQL query, such as UPDATE User SET userName = "Alice" WHERE ..., which updates the user name in the DBMS.

Due to their intuitive abstraction of database access, ORM frameworks are widely used in database-backed web applications [4], [5], [6]. Despite the popularity and convenience of ORM frameworks, they may also cause maintenance challenges [1]. ORM automatically generates SQL queries based on various ORM configurations (e.g., the relationship among object types) and the called ORM APIs. As a result, developers do not have direct control over how the SQL queries are generated by ORM. When there are issues with a generated SQL query, developers may have difficulties knowing how the SQL query is generated and where in the code [4], [7], [8].

• *The authors are with the Software PErformance, Analysis, and Reliability (SPEAR) lab, Concordia University, Montreal, Quebec.*
E-mail: w_liu201@encs.concordia.ca, peterc@encs.concordia.ca

Hibernate, which is one of the most popular Java ORM frameworks, provides a mechanism that allows developers to record the generated SQL queries [9], i.e., ORM logs. Such ORM logs comprehensively record every generated SQL query so that developers would know what the generated queries look like. However, even with the recorded SQL queries, it would be difficult to infer how and locate where a given SQL query is generated [5], [10]. Hibernate generates a SQL query by considering all of the ORM configurations (e.g., how objects should be retrieved from the DBMS), how the objects are accessed, and the executed ORM APIs, on one code path. There may be hundreds or even thousands of database accesses in the source code. Thus, simply searching for the query text in the code would not work, and the generated SQL query may change based on the ORM configuration and the executed branch on an execution path.

Prior studies [11], [12], [13], [14], [15], [16], [17], [18] propose information retrieval based bug localization (IRBL) approaches that try to locate buggy files given some software artifacts (e.g., bug reports). IRBL approaches compute and rank the files based on their similarity with the given software artifact, where the files with the highest similarity are more likely to be defect-prone. IRBL approaches provide good indications of where the bugs are given limited information of the bugs [19]. Similarly, the ORM-generated SQL queries may have quality issues that are caused by incorrect or inefficient usage of ORM code/configuration [6], [20], [21].

In this paper, we propose an approach, SLocator, that combines static analysis and information retrieval techniques to locate the origin (i.e., the control flow path, which contains a sequence of method calls) that generates a given SQL query in database-backed web applications. Different from prior studies on database-backed applications [2], [5], [6], [20], which focus on statically detecting issues based on predefined/known anti-patterns in database-backed web applications, SLocator can be used to locate the origin of any given SQL queries. SLocator also complements existing studies, which rely mostly on static analysis, by providing an approach to help analyze the dynamically-generated SQL queries.

SLocator is a hybrid approach that combines both static code analysis and information retrieval techniques for localization. First, SLocator applies static analysis to analyze the ORM configurations in the source code, which specify the mapping between objects and database tables, and how various objects should be retrieved from the DBMS. Then, SLocator statically analyzes each web request handling method and constructs interprocedural control flow graphs. For each path in the control flow graph, SLocator analyzes both the called ORM APIs and the ORM configurations to statically infer the database access (i.e., templated SQL query). Given a SQL query recorded by a DBMS, SLocator pre-processes the query to remove dynamic elements (e.g., dynamically generated values). Finally, SLocator uses cosine and Jaccard distance to find the control flow path for which the inferred database access has the highest similarity ranking with the given SQL query. Different from existing IRBL approaches, SLocator locates the control flow path that generates a given SQL query instead of the file/method that contains the corresponding ORM code. We choose to locate the control flow path because prior studies found that control flow paths provide additional information for locating the root causes of an issue [18], [22], [23], [24]. Moreover, database access issues may not only exist in ORM API calls but may also be related to how the objects are accessed during execution and the corresponding ORM configuration [6], [8], [20].

We implement SLocator for the Java Persistent API (JPA), which is the official ORM API specification for Java. We evaluate SLocator on seven open source database-backed web applications which use the Hibernate ORM framework. SLocator uses DBMS logs (e.g., MySQL logs) as the input. We use DBMS logs instead of ORM logs because DBMS logs are lightweight and commonly used in production to record problematic SQL queries. In contrast, ORM log introduces significant performance overhead [25], [26], [27], as ORM would record every executed SQL query. Since large-scale web applications may execute hundreds of SQL queries per second, such performance overhead makes enabling ORM logs impractical in production. The dataset of SLocator is publicly available [28].

The main contributions of this paper are:

- SLocator is one of the first techniques that combine interprocedural control flow analysis and information retrieval techniques for localization.
- SLocator is able to locate the control flow path that generates a given set of SQL queries with high accuracy (average Top@5 is 88.8%).
- We evaluate SLocator on existing problematic SQL queries (i.e., slow SQL queries) and we find that SLocator can locate where the SQL queries are generated with similarly high accuracy.
- We conduct a study to illustrate how SLocator helps locate slow SQL queries and database deadlocks in studied applications.

In conclusion, our paper proposes a novel approach that is able to locate the control flow path that generates a given SQL query. Our research also illustrates the potential direction of leveraging static code analysis to enhance software artifact/bug localization techniques.

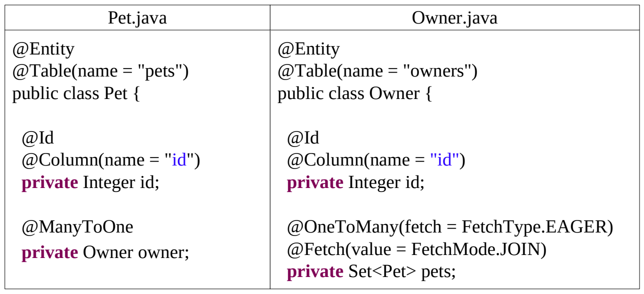
Paper Organization. The rest of the paper is structured as follows. Section 2 introduces the background of using ORM in database-backed web applications and surveys related work. Section 3 presents our approach in detail. Section 4 evaluates our approach on seven open source applications and conducts a study on locating the origin of problematic SQL queries. Section 5 discusses threats to validity. Finally, Section 6 concludes the paper.

2 Background and Related Work

In this section, we first provide some background knowledge of ORM frameworks. Then, we use an example to illustrate the challenge of manually locating the origin of a SQL query (i.e., the control flow path that generates the query). Finally, we discuss related work in two areas: quality assurance of database-backed web applications and IR-based bug localization.

Background of ORM. Object-relational mapping (ORM) frameworks provide a conceptual abstraction between objects in object-oriented languages and the data stored in the underlying DBMS [29]. To leverage ORM frameworks, developers need to first specify ORM configurations. ORM frameworks have two main types of configurations. The first type of the configuration is the mapping configuration, where developers configure the mapping between entity classes and database tables. As shown in Figure 1a, the two entity classes, Pet and Owner (annotated with *@Entity*), are mapped to the pets and owners tables in the DBMS, respectively, using the annotation *@Table*. Both Pet and Owner entities have primary keys (*@Id*) which are mapped to database columns named id (*@Column*). Such mapping configuration allows ORM frameworks to automatically convert an object to/from the corresponding database record.

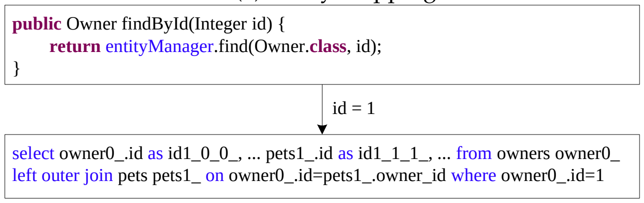
The second type of the configuration is related to entity relationship and data retrieval strategy. ORM provides annotations that allow developers to specify the entity relationship to represent the business logic. For example, Pet has a *@ManyToOne* relationship with Owner, meaning that multiple pets may belong to the same owner. Similarly, there are *@OneToOne*, *@OneToMany*, and *@ManyToMany* relationships. The relationship between entity classes affects how ORM frameworks retrieve the corresponding object from the DBMS. By default, objects with *@OneToOne* and *@ManyToOne* relationship are retrieved together (i.e., eager retrieval), while objects with *@OneToMany* and *@ManyToMany* are not retrieved at the same time (i.e., lazy retrieval) for performance



Pet.java	Owner.java

`@Entity`
`@Table(name = "pets")`
`public class Pet {`
 	
`  @Id`
`  @Column(name = "id")`
`  private Integer id;`
 	
`  @ManyToOne`
`  private Owner owner;`
	`@Entity`
	`@Table(name = "owners")`
	`public class Owner {`
	
	`  @Id`
	`  @Column(name = "id")`
	`  private Integer id;`
	
	`  @OneToMany(fetch = FetchType.EAGER)`
	`  @Fetch(value = FetchMode.JOIN)`
	`  private Set<Pet> pets;`

(a) Entity mapping.



```
public Owner findById(Integer id) {
    return entityManager.find(Owner.class, id);
}
```

id = 1

```
select owner0_.id as id1_0_0_, ... pets1_.id as id1_1_1_, ... from owners owner0_
left outer join pets pets1_ on owner0_.id=pets1_.owner_id where owner0_.id=1
```

(b) Translating objects to SQL queries by ORM.

Fig. 1: An example of accessing the DBMS using ORM.

optimization reasons [30], [31]. Developers can also explicitly specify the retrieval strategy. For example, by adding *FetchType.EAGER* to the configuration of Owner, as shown in Figure 1a, ORM will retrieve all the associated pets at the same time regardless of the type of the relationship. Finally, developers can configure how the associated objects are retrieved. For example, a *FetchMode.JOIN* to the configuration of Owner configures ORM to use an outer join to load the associated pets when fetching.

Despite ORM's advantages in abstracting database access, various configuration options and the paths that the application takes may affect how the SQL query is generated. Therefore, it may cause challenges in locating the origin of the SQL query. For example, as shown in Figure 1b, the ORM API, entityManager.find, retrieves the Owner object from the DBMS based on the owner ID. The corresponding SQL query generated by ORM retrieves not only the owner data but also the associated pets at the same time using an outer join according to the configured ORM annotation. The ORM-generated SQL query does not explicitly exist in the source code and the dynamically generated aliases (e.g., *owner0_* and *pets1_* for the owners and pets tables) introduce discrepancy when locating the origin of a SQL query. Therefore, if there are issues with a generated SQL query, it is challenging to locate where the SQL query is generated when diagnosing the database access code. Moreover, due to the complexity of database-based web applications, different code paths with the same root may generate slightly different SQL queries based on different API calls, which further increases localization difficulty. In short, manually locating where a SQL is generated can be time-consuming and challenging, especially given the size of modern database-backed web applications.

Below, we further discuss related work of this paper.

Quality assurance of database-backed applications. Most prior research aims to study and detect performance issues in database-backed applications that are developed using ORM. Yan et al. [20] studied database-related performance inefficiencies in real-world web applications that are built using the Ruby on Rails ORM framework. They concluded several performance anti-patterns and proposed detection algorithms based on static analysis [2]. Shao et al. [21] presented a comprehensive empirical study that characterizes performance anti-patterns related to database accesses in web applications. Brass and Goldberg [32] summarized common SQL anti-patterns and how to address them. Chen et al. [5], [6] proposed an automated framework to detect and prioritize both performance and functional ORM anti-patterns. Grechanik et al. [33] proposed a run-time monitoring technique to detect database deadlocks.

Most prior studies focus on detecting database access issues at the code level (i.e., anti-patterns). Anti-patterns in ORM code may lead to generating inefficient or incorrect SQL queries. However, one limitation of the prior approaches is that they focus on detecting issues based on predefined/known anti-patterns [21] in database-backed web applications and cannot detect issues that do not belong to any of the predefined anti-patterns. In contrast, given a potentially problematic SQL query (e.g., slow SQL queries or SQL queries that cause database deadlocks), our approach locates the code path that generates the query. Hence, we complement prior approaches by identifying the code that may result in generating problematic SQL queries.

SQL query extracting. Table 1 summarizes the related studies that perform SQL query extracting statically from the source code in their work. The closest related work is by Nagy et al. [10] which is the only study focusing on locating SQL queries. The authors proposed a static concept location approach to match HQL/JPQL query string in the code and the generated SQL query by comparing their abstract syntax trees (AST). However, they did not consider ORM APIs to access entity objects or ORM configurations. In our work, we consider not only static SQL queries (i.e., JPQL) but also ORM API calls to access entity objects and ORM configurations. When using ORM APIs to access entity objects, many SQL queries are generated dynamically, so it is not possible to locate where they are generated by doing string matching. In addition, we locate the control flow path (CFP) that generates a given SQL query instead of the method that contains a database access call (e.g., where the SQL query is defined). Prior studies show that such CFP provides important information when locating a fault and diagnosing the issues [18], [22], [23]. We find that our approach can locate the control flow path that generates a given SQL query with high accuracy.

Other studies on database-backed applications address different issues. Prior studies focus on detecting ORM code smell [34], conducting empirical studies of how SQL queries are constructed [35], extracting SQL queries from source code [36], [37], [38], checking the correctness of SQL queries [39], analyzing SQL queries [40], or detecting SQL anti-patterns [41]. Many studies [35], [37], [39], [40] do not support ORM frameworks and statically extract embedded SQL queries that were manually constructed by developers from the source code. Some studies [34], [36] partially supported ORM frameworks by extracting SQL queries from the source code. However, the extracted SQL queries are different from the dynamically generated SQL queries by the ORM during runtime, which still leaves the task of locating SQL queries challenging.

TABLE 1: Related studies that perform SQL query extracting statically from the source code. ORM, JPQL, and ORM APIs to access entity objects indicate whether the SQL query extracting supports ORM frameworks, JPQL, or ORM APIs to access entity objects, respectively.

Study	Summary of study	Goal of study	ORM	JPQL	ORM APIs to access entity objects
Nagy et al. [10]	They proposed a static concept location approach to match HQL/JPQL query string in the code and the generated SQL query by comparing their abstract syntax trees (AST).	Locating SQL	Yes	Yes	No
Huang et al. [34]	They proposed a static analysis tool, called HBSniff, for detecting 14 code smells.	Detecting ORM code smell	Yes	Yes	No
Anderson [35]	They studied five patterns of SQL query construction in actual PHP systems.	Empirical study of SQL	No	N/A	N/A
Meurice et al. [36]	They presented a static analysis approach to extract SQL queries in Java systems.	Extracting SQL	Yes	Yes	No
Manousis et al. [37]	They presented a method that identifies the embedded queries within database applications.	Extracting SQL	No	N/A	N/A
Nagy and Cleve [38]	They briefly described the tool that is able to extract SQL queries from Java code through static string analysis.	Extracting SQL	No	N/A	N/A
Gould et al. [39]	They presented a static analysis technique for verifying the correctness of dynamically generated SQL query strings for database applications in Java.	Checking SQL	No	N/A	N/A
Annamaa et al. [40]	They described a tool that statically analyzes SQL queries embedded in Java programs.	Analyzing SQL	No	N/A	N/A
Lyu et al. [41]	They proposed a static analysis approach to detect SQL anti-patterns in mobile apps.	Detecting SQL anti-patterns	No	N/A	N/A
Our work	**We proposed an approach to locate the paths that lead to the generated SQL queries.**	**Locating paths for SQL**	**Yes**	**Yes**	**Yes**

Information retrieval based bug localization. Information retrieval based bug localization (IRBL) aims to identify potentially buggy files by computing the similarity between a given software artifact (e.g., bug report) and source code files [13], [15], [42], [43], [44], [45]. The source code files are then ranked based on their similarity with the software artifact for investigation. Zhou et al. [11] proposed an IR-based method named *BugLocator* for locating relevant source code files based on initial bug reports by utilizing a revised Vector Space Model (rVSM) as well as similar bug information. Wong et al. [12] proposed an approach, *BRTracer*, which leverages two techniques segmentation and stack-trace analysis to improve the performance of bug localization. Wang and Lo [14] proposed an approach called *AmaLgam+* that integrates various information (e.g., version history, similar bug reports, and stack traces) to better locate buggy files given a bug report. Lee et al. [16] presented a comprehensive study that compares six state-of-the-art IR-based bug localization techniques. Pradel et al. [17] presented a technique *Scaffle* which uses crash reports to identify the possible file paths and the associated files that may have caused the crash. Chen et al. [18] proposed an IRBL approach, *Pathidea*, which leverages logs in bug reports to re-construct execution paths and they found that the execution path provides a significant improvement in bug localization accuracy. Other works on IRBL focus on optimizing and reformulating queries extracted from the bug report text [46], [47], [48], [49]. Similarly, our approach first applies static analysis to infer the database access (i.e., templated SQL query) for each control flow path. Then, we apply information retrieval (IR) techniques to find the control flow paths for which the inferred database accesses have the highest similarity with the given SQL query. Different from prior IRBL approaches that aim to locate bugs using bug reports, our approach is one of the first to apply IR techniques to locate the origin of SQL queries. Our approach also provides additional information (i.e., the code path) instead of only locating the method that generates the SQL query. Given a problematic SQL query, SLocator can locate the code path that generates the query.

3 Approach

As discussed in Section 2, there are various factors that affect how a SQL query is generated when using ORM. Hence, a simple text search based on the generated SQL query may not be sufficient to locate the origin (i.e., the control flow path, which contains a sequence of method calls) of the SQL query. Figure 2 provides an overview of our approach, SLocator, which automatically locates the origin of the SQL query. SLocator uses a combination of static analysis and information retrieval to locate the path. We first use static analysis to infer the database access of each control flow path in the source code. Then, given SQL queries, we use information retrieval techniques to rank the control flow paths that have the highest database access similarity (i.e., the similarity score between the database access inferred from the control flow path and the given SQL query). We implement SLocator in Java based on the Java Persistent API (JPA), which is Java's official specification for ORM frameworks. Below, we discuss the design of SLocator in detail.

3.1 Statically Inferring Database Access

3.1.1 Generating and Pruning Control Flow Graphs

To locate the possible control flow paths that generate a given SQL query, we use static analysis to construct the interprocedural control flow graph (CFG) of the application [50]. Specifically, the CFG is a directed graph, where the nodes represent the basic blocks and the edges connecting the nodes represent the transfer of control flow between basic blocks. We use Crystal[1], a Java static analysis framework that is built on top of Eclipse JDT, to analyze the source code and construct the CFG.

In database-backed web applications, users often interact with the applications by sending HTTP requests (e.g., using RESTFul APIs or through browsers) [51]. Therefore, SLocator statically analyzes the Java API for RESTful web services (JAX-RS) [52] specifications in the source code to identify a list of web request handling methods. An example of JAX-RS code is shown below:

```
@RequestMapping(value = "/owners/{ownerId}", method =
     RequestMethod.GET)
public Owner showOwner(int ownerId) {
   Owner owner = this.clinicService.findOwnerById(ownerId);
   return owner;
}
```

In this example, based on the JAX-RS annotations, when users send an HTTP GET request that ends with the URL

1. https://code.google.com/archive/p/crystalsaf/

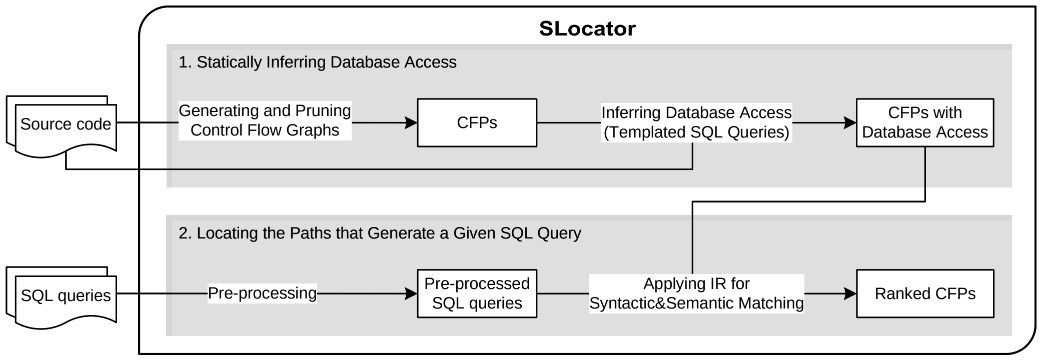

SLocator
1. Statically Inferring Database Access
Source code
Generating and Pruning Control Flow Graphs
CFPs
Inferring Database Access (Templated SQL Queries)
CFPs with Database Access
2. Locating the Paths that Generate a Given SQL Query
SQL queries
Pre-processing
Pre-processed SQL queries
Applying IR for Syntactic&Semantic Matching
Ranked CFPs

Fig. 2: An overview of SLocator. CFP refers to control flow path and IR refers to information retrieval.

"/owners/{ownerId}", method *showOwner* is called to handle the request.

The request handling methods are used as the entry points to the uncovered control flow graphs. For each request handling method, SLocator uncovers all of the associated control flow paths by traversing the interprocedural CFG. As the goal of SLocator is to statically locate the control flow path that results in generating a given SQL query, we omit cycles in the CFG. We perform a depth-first search (DFS) to traverse the CFG and omit the vertex that has been visited before (i.e., a cycle is detected). There may be multiple control flow paths that are associated with one request handling method, and not every path is related to database access. Hence, we further conduct pruning to remove the paths that do not have database access calls. In particular, we analyze if a path contains any API call to the EntityManager class (i.e., the main class in JPA for database accesses). We prune the path if it does not contain any call to EntityManager.

3.1.2 *Statically Inferring the Database Access of Each Control Flow Path*

When using ORM frameworks, many database accesses are abstracted as ORM API calls. As shown in Section 2, in most cases, developers only need to specify the association among classes (e.g., *OneToMany* or *OneToOne* relationships) and different database access configurations (e.g., *EAGER* or *LAZY*). Then, developers can access the DBMS by calling APIs such as EntityManager.find(User.class, userID). Therefore, to statically infer the database access (i.e., templated SQL queries), we analyze both the database access methods that are called on the control flow path and the corresponding ORM configuration. To infer the database access, we implement a database access translator that takes as input the tuple {database access method, entity mapping, association, retrieval strategy}, which are defined as:

- Database access methods: API calls to EntityManager.
- Entity Mapping: Annotations, such as *@Table* and *@Column*, which map an entity class to its corresponding database table.
- Associations: *ManyToOne*, *OneToMany*, *OneToOne*, and *ManyToMany*.
- Retrieval Strategy: *EAGER* or *LAZY*.

Table 2 shows the inferred database accesses given the database access APIs, entity mapping, association, and retrieval strategy. In addition to using method calls such as EntityManager.find(), JPA also provides native SQL to query database tables and JPA query language (JPQL) [31] to create queries against entities. Native SQL queries can be used in the method createNativeQuery(String queryString) while JPQL queries can be used in createQuery(String queryString), where queryString is the SQL query statement and JPQL query statement, respectively, to be executed. During our analysis, we analyze the abstract syntax tree (AST) of the program to extract the potential value of the string variable (i.e., queryString) as the inferred database access (i.e., inferred queries). During the static analysis of the source code, we first use Eclipse JDT to create the AST for the method which contains database access API calls. Then, we handle the argument of the database access API call such as createQuery(String queryString). If the argument is a literal text, we extract it directly. If the argument is a variable, we try to extract its value in the AST based on the prior variable assignment. For the method createNamedQuery(String name) which supports both native SQL and JPQL, the inferred database access queryString is the corresponding named query (i.e., native SQL query or JPQL query) based on the name.

For direct calls to EntityManager APIs, the translator translates the Create, Read, Update, and Delete (CRUD) operations to the corresponding SQL queries using the SQL query templates shown in the table. For each operation, the translator inputs the parameters in the template, such as *table_name*, *column_name*, and *primary_key_name*, based on the entity mapping to generate the inferred SQL queries. When the association is *ManyToOne*, *OneToOne*, or when the retrieval strategy is *EAGER*, the inferred SQL queries for the EntityManager API calls would contain a *join* clause that select data from two or more database tables [6], [31]. For JPQL, the inferred queries would contain multiple select statements to select the data records from the associated tables [53].

3.2 Locating the Paths that Generate a Given SQL Query

SLocator uses information retrieval techniques to locate the origin of a given SQL query. The SQL queries are used as the search term, and the corpus (i.e., collections of documents) are the inferred database accesses. Each document represents the inferred database access, with a mapping to the corresponding control flow path (as discussed in Section 3.1). SLocator compares both the syntactic and semantic similarity between the inferred database access and the given SQL query. SLocator returns a ranked list of the control flow paths

TABLE 2: Translations from ORM API calls to inferred database accesses (templated SQL queries). For native SQL and JPQL, SQL statements or JPQL statements in queryString are extracted as inferred database accesses (inferred queries). Values in { } are statically inferred based on the entity mapping.

	Operation	Database Access API	Inferred Database Access	Inferred Database Access with EAGER retrieval
JPA Entity Manager	Create	persist(Object entity)	insert into {table_name} ({column_name}, ...) values(?, ...)	
	Read	find(Class entityClass, Object primaryKey)	select {column_name} ... from {table_name} where {primary_key_name}=?	select column_name ... from table_name [**join on {table_name.column_name} = {target_table_name.join_column_name}**] * where {primary_key_name}=?
	Update	merge(T entity)	update {table_name} set {column_name}=? ... where {primary_key_name}=?	
	Delete	remove(Object entity)	delete from {table_name} where {primary_key_name}=?	
Native SQL	CRUD	createNativeQuery(String queryString) createNamedQuery(String name)	queryString	
JPA Query Language	CRUD	createQuery(String queryString) createNamedQuery(String name)	queryString	queryString [**select {column_name} ... from {target_table_name} where {primary_key_name}=?**] *

whose inferred database accesses have the highest similarity with the SQL query. Below, we discuss the approach in detail.

3.2.1 Pre-processing SQL Queries

The SQL queries generated by the ORM frameworks may contain dynamic elements (e.g., aliases) that can affect localization accuracy. For example, SQL queries may contain dynamic values that cannot be found in the source code. Consider a SQL query from PetCinic that is generated by Hibernate:

```
select owner0_.id as id1_0_0_, ... pets1_.id as id1_1_1_,
    ... from owners owner0_ left outer join pets pets1_ on
    owner0_.id=pets1_.owner_id where owner0_.id=1
```

where *owner0_* and *pets1_* are aliases for the owners and pets tables, *id1_0_0_* and *id1_1_1_* are aliases for the ID columns in the selected tables. Including such automatically-generated IDs and dynamic values/aliases will reduce the localization accuracy because they do not exist in the inferred database accesses (i.e., templated SQL queries). We pre-process the SQL queries by following pre-processing techniques that are used for software artifacts [42], [43], [44], [45], [54], [55], [56]. We first parse the SQL queries into abstract syntax trees (ASTs) and traverse the ASTs to remove automatically-generated variable and column names, and aliases. Then, we remove the dynamic values (i.e., string literals and numeric values). Finally, we transform all the words into lowercase. After the pre-processing steps, the above-mentioned SQL query becomes:

```
select from owners left outer join pets where owner.id=?
```

Once the SQL queries are pre-processed, we apply information retrieval to find the corresponding inferred database accesses that have the highest similarity.

3.2.2 Applying Information Retrieval for Syntactic and Semantic Matching

Given a SQL query (or a set of SQL queries), we want to find the inferred database accesses that have the highest similarity. In particular, SLocator compares both the syntactic and semantic similarity between the pre-processed SQL queries and the inferred database accesses.

Computing Syntactic Similarity. To compute the syntactic similarity, we represent both the pre-processed SQL query and the inferred database access as strings and calculate the similarity score [55], [57]. Given a SQL query q, SLocator computes the syntactic similarity as the cosine similarity between q and the inferred database access of a control flow path p as follows:

$$sim_{\text{syn}}(q,p) = cosine(\vec{q},\vec{p}) = \frac{\vec{q}\cdot\vec{p}}{\|\vec{q}\|\cdot\|\vec{p}\|}, \tag{1}$$

where $\vec{q}$ and $\vec{p}$ are the weight vectors for the SQL query q and the inferred database access of a control flow path p, respectively. We compute the weight vectors based on the term frequency and inverse document frequency (i.e., $tf \cdot idf$), where more weights are given to words that have higher occurrences in a given document but have lower occurrences in the corpus (i.e., words that are more relevant).

Computing Semantic Similarity. As found in prior studies [55], [58], semantic information in SQL queries such as the accessed tables and operations on tables (e.g., select and update) are useful in identifying similar SQL queries. SLocator uses the Jaccard similarity index to compute the semantic similarity between a SQL query q and the inferred database access of a control flow path p as follows:

$$sim_{\text{sem}}(q,p) = \frac{|features(q) \cap features(p)|}{|features(q) \cup features(p)|}, \tag{2}$$

where *features(p)* are the set of accessed tables and CRUD operations on tables in p. Intuitively, if the accessed tables and operations are different between p and q, it is less likely that the two database accesses are similar.

Combining Similarity Scores and Deriving Path Ranking. We combine the semantic and syntactic similarity to measure the similarity score between a SQL query q and an inferred database access of a path p as follows:

$$Score(q,p) = sim_{\text{syn}}(q,p) + sim_{\text{sem}}(q,p) \tag{3}$$

$Score(q,p)$ ranges between 0 to 2, where the larger the value the higher the similarity. Given q, we compute the $Score(q,p_i)$ for every control flow path p_i generated by the previous steps in our approach. The p_i with a higher similarity score would be ranked higher in the result and are more likely to be the path that generates q.

4 Evaluation

In this section, we first introduce the studied applications and experimental setup. Then, we evaluate SLocator by

TABLE 3: An overview of the studied applications. DB access refers to database access.

Application	Version	LOC	No. of commits	No. of tables	No. of Java files	No. of distinct DB accesses
PetClinic	1.5	2.4K	707	7	38	12
CloudStore	2.0	11.2K	200	11	98	40
WallRide	1.0.0.M18	32.6K	744	35	363	93
JeeWeb	1.0	40.8K	64	31	419	112
PublicCMS	4.0	47.3K	1,103	43	496	132
bbs	5.6	129K	40	44	579	148
Broadleaf-Commerce	6.0.11-GA	197K	17,599	60	1,284	58
Avg. across applications	–	65.8K	2,922	33	463	85

answering two research questions (RQs). For each RQ, we discuss the motivation, approach, and results.

4.1 Evaluation Setup

Studied Applications. We conduct our study on seven open source applications that are popular (i.e., with an average of 1.4K stars on GitHub), have long development history, or used in prior studies on database-backed applications [6], [59], [60], [61], [62], [63], [64]. Table 3 shows an overview of the studied applications, such as the number of commits, database tables, Java files, and distinct database accesses. On average, there are 33 database tables, 463 Java source code files, and 85 distinct database accesses where the SQL queries may be generated. The database-backed web applications are implemented in Java using JPA to access the database. Among all the 595 database accesses, 113 (19.0%) use JPA API persist(), 59 (9.9%) use JPA API find(), 54 (9.1%) use JPA API merge(), 67 (11.3%) use JPA API remove(), 291 (48.9%) use JPQL queries, and 11 (1.8%) use JPA criteria (the statistics of JPA API usage can be found in the online appendix [28]). No native SQL queries are used in the studied applications. Hence, given a large number of database access calls, manual analysis of the origin of a SQL query can be difficult. PetClinic [65] is developed and maintained by Pivotal Software for showcasing standard practices on developing database-backed web applications. CloudStore [66] is an e-commerce web application that is developed according to the TPC-W benchmark [67] while BroadleafCommerce [68] is an enterprise e-commerce framework. Since Broadleaf is a framework, we study the site module provided by BroadleafCommerce, which uses Broadleaf's APIs to build an online shopping website. PublicCMS [69] and WallRide [70] are content management systems (CMSs). JeeWeb [71] is a development system that helps developers generate source code. bbs [72] is a forum application and we study the admin module that is used to manage the forum. In particular, PublicCMS is developed/maintained by a company, has over 1.6K stars on GitHub, is used in many commercial settings, and has many users around the world. BroadleafCommerce has been developed since 2009 and has over 1.5K stars on GitHub.

Experimental Setup. We deploy the studied applications on Tomcat 7, using MySQL 5.6 as the database management system. To simulate a real-world deployment setting, we follow a prior study and populate the main database tables to 20,000 records [2]. For the applications that already contain initial data records, we duplicate these records while keeping their association relationships. For the applications that do not have initial data records, we exercise them by simulating user actions to generate data records and populate the databases. To evaluate SLocator, we exercise the applications by running simulated workloads after the data is populated, and record the application execution information. We first analyze the application usage and then use JMeter [73] to automatically send user requests to simulate hundreds of concurrent users. For each user, we set JMeter to generate random values for variables in the request. Hence, given hundreds of concurrent users, each request would be called hundreds of times with random input values.

TABLE 4: Statistics of running SLocator against the studied applications. Time to locate the paths refers to the average time to rank and locate the control flow paths for a given SQL query.

Application	No. of inferred CFPs	Static analysis execution time (s)	Time to locate the paths (ms)
PetClinic	18	13	7
CloudStore	64	48	12
WallRide	487	175	142
JeeWeb	333	102	20
PublicCMS	1,267	318	41
bbs	2,298	162	120
BroadleafCommerce	1,317	371	679
Avg. across applications	826	170	146

The workload covers most of the application web pages by navigating the menu. For PetClinic, the workload covers user actions such as searching and adding/modifying owners' and pets' information. For CloudStore and Broadleaf-Commerce, the workload covers browsing, searching for items, adding items to carts, and checking out. For WallRide and PublicCMS, the workload covers common actions in CMS such as editing user profiles, adding content (e.g., pictures), editing/adding posts, and editing web pages. For JeeWeb, the workload covers editing/adding system content (e.g., user, department, role), configuring the database, and generating source code to query the database tables. For bbs, the workload covers common actions in forums such as writing posts and questions, editing/adding tags for posts and questions, and viewing posts and questions. Overall, the workload covers 71.2% of the related web requests, 70.7% of the related database accesses, and 74% of the related database tables. Both the workload and SQL queries generated by the workload are publicly available (the statistics of the workload and SQL queries can be found in the online appendix) [28].

Statistics of SLocator. Table 4 shows the statistics of running SLocator against the studied applications. On average, there are 826 control flow paths that contain database access calls leading to generating SQL queries. Note that, each database access may generate multiple SQL queries based on the ORM configuration and each control flow path may contain several database accesses. Hence, the number of generated SQL queries would be even larger, which makes manual analysis of the origin of a SQL query more difficult. We conduct all of our experiments on a Windows 10 machine with Intel Core i5 CPU@1.70GHz and 16GB of RAM. Regarding the execution time, SLocator takes 170 seconds, on average, to statically analyze the source code to infer control flow paths with database access details. SLocator takes an average of 146 milliseconds to rank and locate the control flow paths for given SQL queries. For each release of the application, the static analysis only needs to be executed once. Thus, the performance overhead of SLocator is relatively small.

Approaches and Metrics for Evaluating SLocator. In regular usage of SLocator, we would not need any instrumentation. However, to evaluate the localization accuracy of SLocator, we use AspectJ [74] to instrument the application to get the ground truth (i.e., the web request handling methods and the control flow paths that generate the given SQL query). We only apply the instrumentation to get the ground truth. First, we set the configuration of AspectJ to define the pointcuts to match all the methods within the application source code. During the execution of the workloads, for each user request, AspectJ records all the methods that are executed and the corresponding SQL queries that are sent to the DBMS (i.e., MySQL), which represents the dynamic execution path (i.e., the ground truth). Then, given a SQL query, we apply SLocator to find its origin and compare the origin with the ground truth in the evaluation step. For replication purposes, we make our AspectJ configuration publicly available [28].

We define that a dynamic execution path, d, matches with the statically uncovered control flow path, p, if $p \subset d$. Namely, if every method in p appears in d in the same order, we say that p matches with d (i.e., an ordered set). We define the matching using a subset due to two reasons. First, there may be calls to external frameworks in the dynamic execution paths, which may not be captured in the statically uncovered control flow paths. Second, there may be repeated method calls in the dynamic execution path.

Below, we define the information retrieval metrics that we use to evaluate the effectiveness of SLocator when locating the paths that generate the SQL queries.

Top@K. This metric calculates the percentage of the SQL queries whose dynamic execution path matches with one of the top K results, i.e., successfully located.

Precision@K. Given a SQL query, this metric calculates the percentage of the paths that are correctly located within the given top K results.

$$Precision(K) = \frac{\text{\# correctly located paths in top } K}{K} \tag{4}$$

Mean Average Precision (MAP). Given a SQL query, this metric first calculates the average precision (AP) for every path in the ranked paths as follows:

$$AP = \sum_{i=1}^{N} \frac{Precision(i) \times pos(i)}{\text{total \# of correctly located paths}} \tag{5}$$

where N is the number of ranked paths and $pos(i)$ is an indicator function. $pos(i) = 1$ if the i^{th} path correctly matches with the dynamic execution path. Otherwise, $pos(i) = 0$. For computing MAP, we take the average AP of all the given SQL queries.

Mean Reciprocal Rank (MRR). The reciprocal rank for a SQL query is the reciprocal of the position of the first correctly matched path in the ranked results. This metric calculates the mean of the reciprocal ranks across all SQL queries:

$$MRR = \frac{1}{M} \sum_{j=1}^{M} \frac{1}{rank_j} \tag{6}$$

where M is the number of given SQL queries and $rank_j$ means the position of the first correctly matched path in the ranked list for the j^{th} SQL query.

4.2 RQ1: How effectively can SLocator locate the code path that generates a given SQL query?

Motivation. Due to the discrepancy between the application code and the generated SQL queries, locating where a given SQL query is generated can be a challenging task. In this RQ, we evaluate how well SLocator can locate the paths that generated a given list of SQL queries.

Approach. In production settings, developers often only have access to DBMS logs, where DBMS (e.g., MySQL) records the SQL queries that it executes. DBMS logs often record possibly problematic SQL queries for diagnosing database access issues (e.g., slow SQL queries or SQL queries that caused database deadlocks) [75], [76]. We retrieve DBMS logs from MySQL and use such logs as the input to SLocator to evaluate its effectiveness. Note that, as described in Section 4.1, we obtain the dynamic execution paths that generate the SQL queries (i.e., the ground truth) by instrumenting the applications using AspectJ. For every SQL query recorded in the DBMS log, we map it to the corresponding SQL query and dynamic execution path captured by AspectJ.

We evaluate SLocator by using two types of DBMS logs: individual query log and SQL session log. In the individual query log, MySQL records the execution of individual SQL queries. In the SQL session log, MySQL records all the SQL queries that it executes and groups the queries based on sessions (i.e., connections). Listing 1 shows an example of SQL session log from General Query Log [77] in MySQL which has three columns: session ID, command, and argument.

Listing 1: An example of SQL session log.

```
1 8 Query set session transaction read only
2 8 Query SET autocommit=0
3 8 Query select owner0_.id as id1_0_0_, ... from owners owner0_ left outer join
      pets pets1_ on owner0_.id=pets1_.owner_id where owner0_.id=1
4 8 Query select pettype0_.id as id1_3_0_, ... from types pettype0_ where
      pettype0_.id=1
5 8 Query select visits0_.pet_id as pet_id4_1_0_, visits0_.id as id1_6_0_, ...
      from visits visits0_ where visits0_.pet_id=1
6 8 Query commit
```

For instance, 8 is the session ID, "Query" is the command, and the rest are arguments (i.e., actual SQL query). The session starts with *set session transaction read only*, *SET autocommit=0* (Lines 1-2) and ends with *commit* (Line 6). We identify the session based on ID and extract query statements on Lines 3-5 as input for SLocator. Since the SQL queries are executed in the same connection, they reflect that the queries are generated by one sequential execution in the application (i.e., from the same execution path).

We report two levels of granularity in the SQL query localization results: web request and control flow path. In database-backed web applications, most user actions are handled by various web requests. Therefore, identifying the correct web request and the corresponding request handling methods (i.e., the root of the control flow path) that generate a given SQL query provides an important starting point for investigation. We also report the localization results at a finer-grained level, namely, whether SLocator can locate the execution path that generates a given SQL query.

We also compare SLocator with a baseline approach, which applies text search to locate the origin of a given SQL query at the level of web request handling method (the baseline approach does not contain the static analysis component so it cannot locate control flow paths). Given a

SQL query, we build a corresponding query template and search for matching database accesses in the source code. We use query templates instead of directly using the given SQL query because there may be major differences between the generated SQL queries and the database accesses (JPQL or calls to EntityManager) in the source code [31] (e.g., generated SQL queries may have aliases as discussed in Section 2). The query template consists of keywords such as the CRUD operations (e.g., SELECT, UPDATE, INSERT, and DELETE) and the database tables and conditions used in the given SQL query. For calls to ORM APIs (i.e., EntityManager), the query template consists of keywords such as the ORM API calls and entity names inferred from the given SQL query. For example, given a SQL query SELECT * from owners, we infer the ORM API call EntityManager.find() and the entity name Owner. We rank the matching database accesses by calculating the cosine similarity between the query template and the database accesses. Finally, for each matched database access call, we analyze its call hierarchy to find the corresponding web request handling method as the origin of the SQL query.

Results. Table 5 shows the localization results of using SQL session logs. Overall, we find that SLocator has a high Top@K when locating both the web request and the control flow path that generates the SQL queries. When $K = 1$ and $K = 5$, SLocator achieves an average Top@K of 54.0% and 88.8% when locating control flow paths, and 60.2% and 91.7% when locating web requests. The average MAP and MRR are 0.60 and 0.72 when locating control flow paths, and 0.63 and 0.75 when locating web requests. Table 6 shows the localization results of using individual query logs. Compared to the localization results of using SQL session logs, the localization performance is lower when using individual query logs. The average Top@K is 48.3% for control flow paths and 54.4% for web requests when $K = 1$. When $K = 5$, Top@K is 74.7% and 79.6% for control flow paths and web requests, respectively. Correspondingly, the average MAP and MRR are 0.47 and 0.64 when locating control flow paths, and 0.53 and 0.68 when locating web requests.

SLocator has a much better localization result compared to the baseline. The baseline approach achieves an average Top@5, MAP, and MRR of 28.2%, 0.21, and 0.22, respectively, when using SQL session logs. When using individual query logs, the baseline achieves an average Top@5, MAP, and MRR of 18.4%, 0.15, and 0.15, respectively. Compared to the baseline, SLocator improves the Top@5, MAP, and MRR by 225%, 200%, and 241%, respectively, when using SQL session logs. When using individual query logs, the improvement of Top@5, MAP, and MRR is by 333%, 253%, and 353%, respectively.

We find that the decrease in Top@K when using individual query logs is because there may be multiple control flow paths or web requests that can generate the same SQL query. For example, in PetClinic, the application generates a SQL query to select pets' visit information: select from visit where pet_id=1, which may come from six different request handling methods: processFindform, initCreationForm, showOwner, initUpdateOwnerForm, processUpdateOwnerForm, and processUpdateForm. Even for one request handling method showOwner, this SQL query may come from three different paths. In total, the SQL query may come from nine control flow paths. Therefore, the localization accuracy decreases when using individual query logs.

TABLE 5: The localization results when using SQL session logs. Request-Baseline refers to locating the web request using the baseline approach. Request-SLocator and Path-SLocator refer to using SLocator to locate the web request and control flow path, respectively.

Application	Matching	Top@K			MAP	MRR
		K=1	K=3	K=5		
PetClinic	Request-Baseline	9.5%	14.3%	14.3%	0.12	0.12
	Request-SLocator	52.4%	95.2%	95.2%	0.67	0.76
	Path-SLocator	52.4%	95.2%	95.2%	0.67	0.76
CloudStore	Request-Baseline	28.6%	42.9%	42.9%	0.31	0.35
	Request-SLocator	71.4%	85.7%	92.9%	0.79	0.80
	Path-SLocator	57.1%	85.7%	92.9%	0.71	0.73
WallRide	Request-Baseline	11.4%	22.7%	27.3%	0.17	0.18
	Request-SLocator	59.1%	79.5%	95.5%	0.62	0.73
	Path-SLocator	54.5%	77.3%	95.5%	0.65	0.71
JeeWeb	Request-Baseline	4.0%	16.2%	16.2%	0.09	0.09
	Request-SLocator	40.4%	85.9%	90.9%	0.38	0.65
	Path-SLocator	37.4%	78.8%	82.8%	0.35	0.66
PublicCMS	Request-Baseline	10.0%	20.0%	26.0%	0.16	0.17
	Request-SLocator	86.0%	96.0%	98.0%	0.75	0.93
	Path-SLocator	70.0%	88.0%	94.0%	0.67	0.83
bbs	Request-Baseline	31.4%	34.3%	35.7%	0.32	0.33
	Request-SLocator	64.3%	85.7%	91.4%	0.62	0.77
	Path-SLocator	58.6%	78.6%	82.9%	0.57	0.73
Broadleaf-Commerce	Request-Baseline	26.1%	34.8%	34.8%	0.30	0.30
	Request-SLocator	47.8%	60.9%	78.3%	0.59	0.60
	Path-SLocator	47.8%	60.9%	78.3%	0.56	0.60
Avg. across applications	Request-Baseline	17.3%	26.5%	28.2%	0.21	0.22
	Request-SLocator	60.2%	84.1%	91.7%	0.63	0.75
	Path-SLocator	54.0%	80.6%	88.8%	0.60	0.72

In contrast, there may be fewer web requests and control flow paths that generate a given SQL session log. SQL queries in the SQL session log are more likely generated by the same business logic (e.g., the same web request). Hence, it is less likely that multiple web requests and control flow paths generate the same set of SQL queries. For example, in PetClinic, the SQL session log shown in the previous example in Listing 1 may come from only two request handling methods (i.e., showOwner and initUpdateOwnerForm) and four control flow paths. However, if we consider the individual SQL query on Line 5, it may be generated by six request handling methods and nine control flow paths. Nevertheless, our findings show that SLocator can still achieve good accuracy when localizing the path given one single SQL query.

We apply SLocator on the seven studied applications and find that, on average, developers need to investigate two control flow paths (the paths have an average of six methods) when using SQL session logs, and five control flow paths (the paths have an average of eight methods) when using individual query logs, to find the origin of the SQL query. Hence, developers do not need to investigate many returned paths to find the correct SQL origin when using SLocator.

Discussion. As shown in Tables 5 and 6, SLocator has a very high Top@K across the studied applications. However, we find that even if we increase K (e.g., set K to 10 or 20), the numbers still may not reach 100%. The finding shows that there may be some SQL queries for which we cannot find the corresponding statically inferred control flow path. After some manual investigation, we find that such mismatches are caused by the limitation of static analysis and the frameworks that these applications use. For example, PetClinic uses the Spring framework [78] for web request handling and adds the *@ModelAttribute*="visit" annotation to the method loadPetWithVisit(). The method loadPetWithVisit() contains a database access call, but the method is not used in

TABLE 6: The localization results when using individual query logs. Request-Baseline refers to locating the web request using the baseline approach. Request-SLocator and Path-SLocator refer to using SLocator to locate the web request and control flow path, respectively.

Application	Matching	Top@K			MAP	MRR
		K=1	K=3	K=5		
PetClinic	Request-Baseline	15.4%	23.1%	23.1%	0.19	0.19
	Request-SLocator	69.2%	92.3%	100.0%	0.65	0.82
	Path-SLocator	69.2%	92.3%	100.0%	0.65	0.82
CloudStore	Request-Baseline	16.1%	19.4%	19.4%	0.18	0.18
	Request-SLocator	61.3%	77.4%	87.1%	0.61	0.71
	Path-SLocator	45.2%	61.3%	80.6%	0.47	0.58
WallRide	Request-Baseline	8.0%	13.6%	15.9%	0.11	0.11
	Request-SLocator	35.2%	54.5%	60.2%	0.36	0.50
	Path-SLocator	30.7%	52.3%	59.1%	0.34	0.47
JeeWeb	Request-Baseline	3.3%	10.6%	10.6%	0.06	0.06
	Request-SLocator	52.0%	87.0%	90.2%	0.46	0.72
	Path-SLocator	48.8%	78.0%	80.5%	0.42	0.74
PublicCMS	Request-Baseline	10.8%	18.9%	18.9%	0.14	0.14
	Request-SLocator	62.2%	71.6%	73.0%	0.55	0.74
	Path-SLocator	52.7%	66.2%	68.9%	0.47	0.66
bbs	Request-Baseline	26.0%	30.0%	30.0%	0.28	0.28
	Request-SLocator	53.0%	69.0%	77.0%	0.52	0.68
	Path-SLocator	48.0%	63.0%	71.0%	0.48	0.68
Broadleaf-Commerce	Request-Baseline	8.7%	10.9%	10.9%	0.10	0.10
	Request-SLocator	47.8%	63.0%	69.6%	0.54	0.59
	Path-SLocator	43.5%	54.3%	63.0%	0.44	0.53
Avg. across applications	Request-Baseline	12.6%	18.1%	18.4%	0.15	0.15
	Request-SLocator	54.4%	73.5%	79.6%	0.53	0.68
	Path-SLocator	48.3%	66.8%	74.7%	0.47	0.64

all the Java files. We find that the model attribute (i.e., "visit") is referenced in one of the JSP (Java Server Page) files that takes input from the user. In other words, loadPetWithVisit() is called automatically when a user submits a web form to the application server, which is the reason why SLocator was not able to find the control flow path that generates the given SQL query. The issue is common across the studied applications. As another example, in WallRide, developers override the method postHandle() from the Spring framework. postHandle() is executed automatically after handling each web request, and the overridden postHandle() contains database access calls.

In short, even though our results show that SLocator is able to locate the path where a given SQL query is generated with good accuracy, there are still some limitations caused by static analysis. Future studies may consider the frameworks that the application uses to increase the accuracy of static analysis.

We find that SLocator achieves good localization accuracy. When using SQL session logs, the origin (i.e., the CFP) of 54% of the SQL queries can be located in Top@1, and almost 89% can be located in Top@5. When using individual SQL queries, the origin of more than 48% of the SQL queries can be located in Top@1, and almost 75% can be located in Top@5. On average, developers need to investigate two and five control flow paths to find the origin when using SQL session logs and individual SQL queries, respectively.

4.3 RQ2: What is the localization accuracy for SQL queries with different lengths?

Motivation. In RQ1, we evaluate the overall localization accuracy of SLocator. However, the recorded SQL queries may have different complexities such as lengths which may affect how SLocator performs. In this RQ, we evaluate the accuracy of SLocator in localizing the paths for SQL queries with different lengths (i.e., the number of words involved).

Approach. The goal of RQ2 is to assess the ability of SLocator to locate SQL queries with different lengths. Since the range of SQL query lengths varies in the studied applications, instead of using a pre-defined threshold for all the applications, we classify the length of the SQL queries in each studied application into three buckets based on the quantiles (i.e., bottom $1/3$, middle $1/3$, and top $1/3$). We use length (number of words) as a proxy for the complexity of a SQL query, whereas a longer SQL query has a higher complexity. We evaluate the effectiveness of SLocator on localizing the paths for SQL queries in each studied application across the three length groups.

Results. Table 7 shows the localization results for SQL queries at different length groups (i.e., bottom $1/3$, middle $1/3$, and top $1/3$) when using individual query logs. We observe that, in most studied applications, bottom-length SQL queries get better localization results than middle-length SQL queries, which in turn get better localization results than top-length SQL queries. When locating web requests for SQL queries that belong to the three length groups, the average Top@1 are 65.2%, 51.4%, and 42.7% while the Top@5 are 88.5%, 75.1%, and 73.2%, respectively. When locating control flow paths for SQL queries that belong to the three length groups, the average Top@1 are 59.6%, 45.2%, and 40.3% while the Top@5 are 86.1%, 69.6%, and 70.5%, respectively. We find that the decrease in Top@K for SQL queries at different length groups is because longer SQL queries have more words involved, and therefore, are harder to be matched with corresponding control flow paths' inferred database accesses compared to shorter SQL queries. Nevertheless, SLocator achieves good localization results for SQL queries with different lengths. For the SQL queries with length in the top $1/3$, the average Top@5 are 73.2% and 70.5% for web requests and control flow paths, respectively. Compared to the result in RQ1, the decreases are by 8% and 6%, respectively. These results suggest that SLocator can be used to effectively locate the code path for SQL queries at different length groups.

We find that SLocator achieves better localization results for short SQL queries compared to long SQL queries. For the SQL queries with length in the top $1/3$, SLocator achieves good localization results - the average Top@5 are 73.2% and 70.5% for web requests and control flow paths, respectively.

4.4 RQ3: Can SLocator help localize issues in database-backed web applications?

Motivation. Database access performance is critical in database-backed applications since it directly affects the user-perceived quality [2], [6], [20]. Most DBMSs record slow SQL queries and database deadlocks for developers to conduct further investigation [79], [80], [81]. Slow SQL log records the SQL queries that take longer than a predefined threshold (e.g., one second) to execute. Such slow SQL queries may indicate performance issues or opportunities for performance optimization. Deadlock log records the SQL queries that were blocked when deadlocks happen. In database-backed

TABLE 7: The localization results for SQL queries with different lengths (i.e., bottom, middle, and top) when using individual query logs. The length of SQL queries is measured using the number of words and is classified into three buckets based on the quantiles (i.e., bottom 1/3, middle 1/3, and top 1/3). Request and Path refer to using SLocator to locate the web request and control flow path, respectively. SQL lengths refer to the range of SQL query lengths.

Application	Matching	Bottom length						Middle length						Top length					
		SQL lengths	Top@K K=1	Top@K K=3	Top@K K=5	MAP	MRR	SQL lengths	Top@K K=1	Top@K K=3	Top@K K=5	MAP	MRR	SQL lengths	Top@K K=1	Top@K K=3	Top@K K=5	MAP	MRR
PetClinic	Request	7–9	75.0%	100.0%	100.0%	0.58	0.88	10–31	75.0%	100.0%	100.0%	0.71	0.88	59–113	60.0%	80.0%	100.0%	0.66	0.74
	Path		75.0%	100.0%	100.0%	0.58	0.88		75.0%	100.0%	100.0%	0.71	0.88		60.0%	80.0%	100.0%	0.66	0.74
CloudStore	Request	5–20	80.0%	90.0%	90.0%	0.69	0.84	21–68	60.0%	80.0%	90.0%	0.60	0.73	79–266	45.5%	63.6%	81.8%	0.55	0.58
	Path		60.0%	60.0%	90.0%	0.44	0.67		40.0%	70.0%	80.0%	0.50	0.57		36.4%	54.5%	72.7%	0.48	0.50
WallRide	Request	5–12	51.7%	72.4%	86.2%	0.51	0.64	12–57	24.1%	44.8%	44.8%	0.25	0.34	57–990	30.0%	46.7%	50.0%	0.35	0.42
	Path		44.8%	69.0%	82.8%	0.48	0.60		20.7%	44.8%	44.8%	0.24	0.32		26.7%	43.3%	50.0%	0.33	0.40
JeeWeb	Request	5–10	65.9%	95.1%	95.1%	0.52	0.81	10–51	26.8%	85.4%	90.2%	0.36	0.57	51–141	63.4%	80.5%	85.4%	0.50	0.74
	Path		63.4%	87.8%	87.8%	0.49	0.76		24.4%	75.6%	78.0%	0.31	0.50		58.5%	70.7%	75.6%	0.46	0.67
PublicCMS	Request	5–6	75.0%	83.3%	87.5%	0.70	0.80	6–22	64.0%	72.0%	72.0%	0.54	0.69	23–106	48.0%	64.0%	64.0%	0.52	0.56
	Path		70.8%	83.3%	87.5%	0.60	0.78		56.0%	68.0%	68.0%	0.53	0.63		48.0%	64.0%	64.0%	0.46	0.56
bbs	Request	5–7	75.8%	84.8%	93.9%	0.71	0.82	7–21	69.7%	81.8%	81.8%	0.57	0.75	22–100	14.7%	41.2%	55.9%	0.28	0.33
	Path		69.7%	78.8%	87.9%	0.65	0.76		60.6%	69.7%	69.7%	0.53	0.65		14.7%	41.2%	55.9%	0.27	0.33
Broadleaf-Commerce	Request	5–11	33.3%	46.7%	66.7%	0.49	0.45	13–39	40.0%	46.7%	46.7%	0.46	0.47	39–447	37.5%	62.5%	75.0%	0.46	0.52
	Path		33.3%	46.7%	66.7%	0.46	0.45		40.0%	40.0%	46.7%	0.36	0.45		37.5%	62.5%	75.0%	0.45	0.52
Avg. across applications	Request	–	65.2%	81.8%	88.5%	0.60	0.75	–	51.4%	73.0%	75.1%	0.50	0.63	–	42.7%	62.6%	73.2%	0.47	0.56
	Path	–	59.6%	75.1%	86.1%	0.53	0.70	–	45.2%	66.9%	69.6%	0.45	0.57	–	40.3%	59.5%	70.5%	0.44	0.53

applications, each database transaction may execute multiple SQL queries. Deadlocks happen when two or more database transactions are waiting for one another to give up locks. Database deadlock is one of the main reasons for major performance degradation [5], [33].

In the previous RQs, we evaluate the overall localization accuracy of SLocator. In this RQ, we conduct two case studies, i.e., slow SQL queries and SQL queries that cause database deadlocks, to illustrate how SLocator helps localize database access issues in database-backed web applications.

Approach. As described in Section 4.1, we populate the data in the database since many performance issues only occur under large loads [6], [20]. We evaluate SLocator using either existing or injected slow SQL queries or database deadlocks. Below, we discuss how we trigger/inject the performance issues.

Triggering Slow Queries: To trigger slow queries, we exercise the applications by running the same workload that we used in RQ1 and configure MySQL to record the execution time of each SQL query. Then, we calculate the average execution time for each unique SQL query and take the top 10% most time-consuming queries as slow SQL queries by following a prior study [2].

Injecting and Triggering Database Deadlocks: Injecting deadlocks requires much manual effort, and a deep understanding of the database tables and the business logic of the system. Therefore, we choose WallRide, a medium size application with 35 database tables, to inject a deadlock. The size of WallRide is not too small for a study on deadlocks and is still feasible for us to manually study the application source code to inject a deadlock. However, since SLocator achieves similar localization results across the studied applications, we believe SLocator can still help locate the origin of deadlocking SQL queries in other applications. We inject a deadlock in WallRide by changing the lock model type from *PESSIMISTIC_WRITE* (i.e., pessimistic write lock) to *NONE* (i.e., no lock) [82] (Lines 3-4), as shown in Listing 2.

The method *PostRepositoryImpl.lock* is called before retrieving data from the DBMS and locks the corresponding database records with a pessimistic write lock. A pessimistic write lock is an exclusive lock in MySQL [83], which prevents concurrent writing of the same records and reduces the likelihood of deadlock in the database. By changing the lock to *NONE*, there will be chances that a deadlock may happen. After injecting the deadlock, we build and deploy the modified application. We use JMeter to automatically send user requests and simulate hundreds of concurrent users to trigger the deadlock.

Listing 2: Database deadlock injected in WallRide.

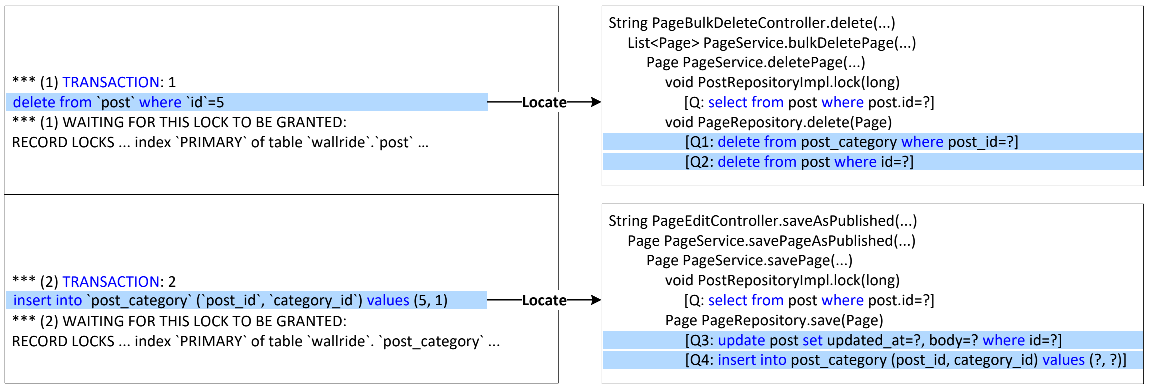
```
1 public void PostRepositoryImpl.lock(long id) {
2 ...
3 -  entityManager.createQuery(query).setLockMode(LockModeType.PESSIMISTIC_WRITE)
       .getSingleResult();
4 +  entityManager.createQuery(query).setLockMode(LockModeType.NONE)
       .getSingleResult();
```

Results.
Slow Queries: We give an example from PetClinic to demonstrate how SLocator locates the origin of a slow SQL query. The slow SQL log identifies the following SQL query as slow in PetClinic:

```
select distinct owner0_.id as id1_0_0_, ... from owners
    owner0_ left outer join pets pets1_ on
    owner0_.id=pets1_.owner_id where owner0_.last_name
    like '%'
```

The SQL query searches for the owner whose last name matches with any string (i.e., like '%', where the wildcard '%' means a string with zero or more characters). By using this SQL query as the input to SLocator, SLocator returns the control flow path as shown in Listing 3 as the first ranked result. To gain more information about the inferred control flow paths, SLocator also returns calls to third-party libraries in the returned path. For instance, the request method processFindForm calls the methods findOwnerByLastName and java.util.Map.put, while the generic method java.util.Map.put is from Java's util library. The method findByLastName accesses the DBMS and generates three SQL queries Q1, Q2, and Q3, where the slow query is generated (i.e., Q1).

Listing 4 shows the corresponding source code containing the origin of the slow SQL query. Based on the control flow path returned by SLocator, the potential execution path of the source code covers Line 3 and Lines 9-10 (as highlighted in



```
*** (1) TRANSACTION: 1
delete from `post` where `id`=5
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS ... index `PRIMARY` of table `wallride`.`post` ...
```

```
*** (2) TRANSACTION: 2
insert into `post_category` (`post_id`, `category_id`) values (5, 1)
*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS ... index `PRIMARY` of table `wallride`. `post_category` ...
```

```
String PageBulkDeleteController.delete(...)
    List<Page> PageService.bulkDeletePage(...)
        Page PageService.deletePage(...)
            void PostRepositoryImpl.lock(long)
                [Q: select from post where post.id=?]
            void PageRepository.delete(Page)
                [Q1: delete from post_category where post_id=?]
                [Q2: delete from post where id=?]
```

```
String PageEditController.saveAsPublished(...)
    Page PageService.savePageAsPublished(...)
        Page PageService.savePage(...)
            void PostRepositoryImpl.lock(long)
                [Q: select from post where post.id=?]
            Page PageRepository.save(Page)
                [Q3: update post set updated_at=?, body=? where id=?]
                [Q4: insert into post_category (post_id, category_id) values (?, ?)]
```

(a) Deadlock log obtained from MySQL. A deadlock happens when two transactions T1 and T2 access database tables post and post_category.

(b) The control flow paths and inferred database access for the deadlock SQL queries.

Fig. 3: Using SLocator to locate the paths in the control flow graphs that result in generating deadlock SQL queries.

blue). Line 10 indicates that all the owners retrieved from the DBMS will be displayed on one web page ownersList.html. The performance issue occurs when there are many owners whose last name matches the given search string. For this particular SQL query, it retrieves and displays all the users. A solution is to add pagination so that only a limited number of owners would be retrieved and displayed for every page. Note that, in this example, the performance issue would not occur if the code executes the first or the second branch (i.e., the number of matched owners is zero or one). Hence, the control flow path that is returned by SLocator may provide additional information to locate performance issues.

Listing 3: The control flow path and inferred database access returned by SLocator for a slow SQL query in PetClinic.

```
String ownercontroller.processFindForm(Owner, BindingResult, Map)
   Collection<Owner> ClinicServiceImpl.findOwnerByLastName(String)
      Collection<Owner> JPAOwnerRepositoryImpl.findByLastName(String)
         [Q1: select distinct owner from owner owner left join fetch
               owner.pets where owner.lastname like :lastname]
         [Q2: select id, name from types where id=?]
         [Q3: select id, visit_date, description from visits where id=?]
   V java.util.Map.put(K, V)
```

Listing 4: Source code containing the origin of the slow SQL query.

```
1  String ownercontroller.processFindForm(Owner, BindingResult, Map){
2     // find owners by last name
3     Collection<Owner> results =
         this.clinicService.findOwnerByLastName(owner.getLastName());
4     if (results.isEmpty()) {           // branch 1: no owners found
5        ...
6     } else if (results.size() == 1) { // branch 2: 1 owner found
7        ...
8     } else {                           // branch 3: multiple owners found
9        model.put("selections", results);
10       return "owners/ownersList";
11    }
12 }
```

Database Deadlocks: We use the injected deadlock in WallRide to illustrate the usage of SLocator to locate the origin of deadlock SQL queries for further diagnosis. Figure 3a shows the deadlock log obtained by using the MySQL command SHOW ENGINE INNODB STATUS. Two SQL queries are blocked (as highlighted in blue), waiting for a lock to be granted in transactions T1 (TRANSACTION: 1) and T2 (TRANSACTION: 2), respectively. However, it is unknown how this deadlock happens according to the log because these two SQL queries should not block each other as they access different tables (i.e., post and post_category).

By using the first blocked SQL query in transaction T1 as the input to SLocator, SLocator returns the first control flow path shown in Figure 3b. Note that the path returned by SLocator is one specific path in the control flow graph, so the methods are on the same call path (i.e., no branching in between). The request handling method PageBulkDeleteController.delete (i.e., the root of the returned control flow path) calls the method PageService.bulkDeletePage, which then calls the method PageService.deletePage. PostRepositoryImpl.lock accesses the DBMS and generates the SQL queries Q. PageRepository.delete accesses the DBMS and generates two SQL queries Q1 and Q2, where SLocator locates Q2 as where the first blocked SQL query is generated. Note that Q and Q1 must have been executed since Q, Q1, and Q2 are on the same control flow path. Similarly, using the second blocked SQL query in Figure 3a as the input, SLocator returns the second control flow path shown in Figure 3b.

Based on the control flow paths that are returned by SLocator, we can see that Q1 and Q4 access the post_category table, and Q2 and Q3 access the post table. Since Q1 and Q4, and Q2 and Q3 are in different methods, a deadlock may happen when the two methods are executed by two separate transactions. For example, a transaction T1 may hold the lock on the post_category table (i.e., executing Q1) while another transaction T2 holds the lock on the post table (i.e., executing Q3). In this case, T1 cannot execute Q2 because T2 is holding the lock; and T2 cannot execute Q4 because T1 is holding the lock. Possible ways to solve this deadlock are to execute the SQL queries that access the same set of tables in a fixed order, or add a pessimistic lock (as shown in Listing 2). In short, the origins of the SQL queries that are returned by SLocator may provide developers additional information to investigate the root cause of deadlocks.

We evaluate SLocator to illustrate its usage in locating the application code that results in generating two cases of problematic SQL queries, i.e., slow SQL queries and SQL queries that cause database deadlocks. We find that SLocator provides developers with additional information to localize the database access issues.

5 Threats to Validity

External Validity. We only evaluate SLocator on seven open source applications implemented using the Hibernate ORM, which may affect the generalizability of our results. To mitigate these threats, we choose the studied applications with various sizes (the LOC ranges from 2.4K to 197K) across different domains such as e-commerce, CMS, and forum to improve the generalizability. Another threat may come from the studied application PetClinic which only has 2.4K lines of source code. However, we find that the average accuracy does not change much after excluding PetClinic. For example, when using SQL session logs, the average Top@5 for locating the web request changes from 91.7% to 91.2% while the average Top@5 for locating the control flow path changes from 88.8% to 87.7%. Besides, the approach in SLocator may be applicable to applications using non-Hibernate ORM. Future studies may apply the needed changes to evaluate how SLocator performs on applications implemented using other ORM frameworks.

Construct Validity. One possible threat to construct validity might come from the workload in our experimental setup. We use simulated workload to exercise different workflows in the studied applications. However, this simulated workload may not cover all the workflows and may not be representative of real application workflows. To mitigate these threats, our workload achieves high coverage of the web page and database table, although a high value in different metrics is needed to fully address the construct threat.

Limitation of static analysis in our approach. One threat is the limitation of static analysis in inferring control flow paths (as discussed in RQ1). For example, due to different used frameworks and the embedded code logic in the user interface (UI), static analysis may not be able to infer the complete code path that generates a SQL query. Future studies need to consider the peculiarity of various web frameworks of applications when analyzing the source code statically. Another limitation may exist in inferring the database access that occurs outside of the web requests. We choose the web request handling methods as the entry points to the studied applications because most functionalities in a web-based application are accessible through web requests. To verify our design decision, we conducted a backward control flow analysis on the entry points of the database access calls in the studied applications. We found that among all the database accesses, only one database access call in JeeWeb is triggered by a timer in the application instead of accessible through web requests. The limitation of static analysis may also exist in inferring database accesses. Our approach translates the basic and commonly used CRUD operations of the JPA API calls to infer database accesses (templated SQL queries), and extracts the native SQL query and JPQL query as inferred database accesses (inferred queries) (as discussed in Section 3.1.2). We did not include criteria APIs due to their dynamic nature. However, we carefully checked the source code of the seven studied applications and found that criteria queries are less used compared to the basic CRUD operations of the JPA EntityManager (there are only 11 usages of Criteria among all the 595 usages of JPA APIs in the studied applications). Future studies should consider examining the usage of various JPA APIs and may expand SLocator's translation layer to cover APIs such as Criteria.

Populated database. We use the synthesized database content to populate the main database tables (as discussed in Section 4.1). However, the applications running on the synthesized data may behave differently from the actual deployments, which may affect the execution of the studied applications. To mitigate these threats, we try to populate realistic values into the database. For example, we populate unique email addresses and realistic addresses into customer and address tables in BroadleafCommerce. Besides, all of the synthesized database data still follows the association relationships and database constraints in the database. Hence, the applications should execute well on the synthesized database data. The synthetic database data and data-populating scripts (written in procedures in MySQL) are publicly available [28].

6 Conclusion

Object-relational mapping (ORM) frameworks are widely used to abstract database access in database-backed web applications. However, when using ORM, developers do not have full control of how a SQL query is generated. Therefore, given a problematic SQL query, developers may encounter challenges to know how and locate where the SQL query is generated. In this paper, we propose SLocator, an automated approach to locate the control flow path (i.e., the origin) that generates a given SQL query. SLocator combines static analysis and information retrieval (IR) techniques for locating the origin. We evaluate SLocator on seven open source applications by using two types of DBMS logs: SQL session log and individual query log. SLocator achieves good localization accuracy and has a better localization result compared to the baseline. We also conduct a study to demonstrate how SLocator may be used to locate the database access code that generates problematic SQL queries (i.e., slow SQL queries and database deadlocks). Our findings show the potential of using IR techniques to help locate database-related issues.

References

[1] T.-H. Chen, W. Shang, J. Yang, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, "An empirical study on the practice of maintaining object-relational mapping code in java systems," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16, 2016, pp. 165–176.

[2] J. Yang, P. Subramaniam, S. Lu, C. Yan, and A. Cheung, "How not to structure your database-backed web applications: A study of performance bugs in the wild," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, 2018, pp. 800–810.

[3] (2022) Java development research report: Landscape 2014. [Online]. Available: https://www.jrebel.com/blog/java-tools-and-technologies-landscape-2014

[4] J. Yang, C. Yan, C. Wan, S. Lu, and A. Cheung, "View-centric performance optimization for database-backed web applications," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, 2019, pp. 994–1004.

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

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

[7] V. F. C. (2022) Hibernate debugging - finding the origin of a query. [Online]. Available: https://dzone.com/articles/hibernate-debugging-where-does

[8] 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*, vol. 42, no. 12, pp. 1148–1161, 2016.

[9] "Hibernate show real sql," https://stackoverflow.com/questions/2536829/hibernate-show-real-sql, 2021, last accessed Aug. 2021.

[10] C. Nagy, L. Meurice, and A. Cleve, "Where was this sql query executed? a static concept location approach," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 580–584.

[11] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 14–24.

[12] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 181–190.

[13] X. Xia, D. Lo, X. Wang, C. Zhang, and X. Wang, "Cross-language bug localization," in *Proceedings of the 22nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 275–278.

[14] S. Wang and D. Lo, "Amalgam+: Composing rich information sources for accurate bug localization," *J. Softw. Evol. Process*, vol. 28, no. 10, pp. 921–942, Oct. 2016.

[15] M. Wen, R. Wu, and S. Cheung, "Locus: Locating bugs from software changes," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 262–273.

[16] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. Le Traon, "Bench4bl: Reproducibility study on the performance of ir-based bug localization," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 61–72.

[17] M. Pradel, V. Murali, R. Qian, M. Machalica, E. Meijer, and S. Chandra, "Scaffle: Bug localization on millions of files," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020, 2020, pp. 225–236.

[18] A. R. Chen, T.-H. P. Chen, and S. Wang, "Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[19] V. Murali, L. Gross, R. Qian, and S. Chandra, "Industry-scale ir-based bug localization: A perspective from facebook," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 188–197.

[20] C. Yan, A. Cheung, J. Yang, and S. Lu, "Understanding database performance inefficiencies in real-world web applications," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, ser. CIKM '17, 2017, pp. 1299–1308.

[21] S. Shao, Z. Qiu, X. Yu, W. Yang, G. Jin, T. Xie, and X. Wu, "Database-access performance antipatterns in database-backed web applications," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 58–69.

[22] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: Association for Computing Machinery, 2008, pp. 308–318.

[23] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: Error diagnosis by connecting clues from run-time logs," *SIGPLAN Not.*, vol. 45, no. 3, pp. 143–154, Mar. 2010.

[24] M. Soltani, F. Hermans, and T. Bäck, "The significance of bug report elements," *Empirical Software Engineering*, vol. 25, pp. 1–40, 11 2020.

[25] Hibernate, "Logging," 2021, last accessed Jul. 2021. [Online]. Available: https://docs.jboss.org/hibernate/orm/5.3/userguide/html_single/Hibernate_User_Guide.html#best-practices-logging

[26] L. Zeng, Y. Xiao, and H. Chen, "Linux auditing: Overhead and adaptation," in *2015 IEEE International Conference on Communications (ICC)*, 2015, pp. 7168–7173.

[27] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, pp. 415–425.

[28] (2021) Online appendix/replication package for "SLocator: Localizing the origin of sql queries in database-backed web applications". [Online]. Available: https://github.com/liuwei-tianshu/SLocator

[29] T.-H. Chen, "Improving the performance of database-centric applications through program analysis," Ph.D. dissertation, Queen's University (Canada), 2016.

[30] Oracle, "Jsr 338: Java persistence api, version 2.2," 2017, last accessed Jul. 2021. [Online]. Available: https://download.oracle.com/otn-pub/jcp/persistence-2_2-mrel-eval-spec/JavaPersistence.pdf

[31] M. Keith, M. Schincariol, and M. Nardone, *Pro JPA 2 in Java EE 8: An In-Depth Guide to Java Persistence APIs*, 3rd ed. Berkeley, CA: Apress, 2018, ch. 4, pp. 101–155.

[32] S. Brass and C. Goldberg, "Semantic errors in sql queries: a quite complete list," in *Proceedings of the 4th International Conference on Quality Software*, ser. QSIC '04, 2004, pp. 250–257.

[33] M. Grechanik, B. M. M. Hossain, U. Buy, and H. Wang, "Preventing database deadlocks in applications," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, pp. 356–366.

[34] Z. Huang, Z. Shao, G. Fan, H. Yu, K. Yang, and Z. Zhou, "Hbsniff: A static analysis tool for java hibernate object-relational mapping code smell detection," *Sci. Comput. Program.*, vol. 217, no. C, may 2022.

[35] D. Anderson, "Modeling and analysis of sql queries in php systems," Master's thesis, East Carolina University, April 2018.

[36] L. Meurice, C. Nagy, and A. Cleve, "Static analysis of dynamic database usage in java systems," in *Advanced Information Systems Engineering*, S. Nurcan, P. Soffer, M. Bajec, and J. Eder, Eds. Cham: Springer International Publishing, 2016, pp. 491–506.

[37] P. Manousis, A. Zarras, P. Vassiliadis, and G. Papastefanatos, "Extraction of embedded queries via static analysis of host code," in *Advanced Information Systems Engineering*, E. Dubois and K. Pohl, Eds. Cham: Springer International Publishing, 2017, pp. 511–526.

[38] C. Nagy and A. Cleve, "Sqlinspect: A static analyzer to inspect database usage in java applications," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 93–96.

[39] C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. USA: IEEE Computer Society, 2004, p. 645–654.

[40] A. Annamaa, A. Breslav, J. Kabanov, and V. Vene, "An interactive tool for analyzing embedded sql queries," in *Programming Languages and Systems*, K. Ueda, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 131–138.

[41] Y. Lyu, S. Volokh, W. G. J. Halfond, and O. Tripp, "Sand: A static analysis approach for detecting sql antipatterns," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 270–282.

[42] T.-D. B. Le, F. Thung, and D. Lo, "Predicting effectiveness of ir-based bug localization techniques," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014, pp. 335–345.

[43] F. Thung, T.-D. B. Le, P. S. Kochhar, and D. Lo, "Buglocalizer: Integrated tool support for bug localization," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 767–770.

[44] S. Wang, D. Lo, and J. Lawall, "Compositional vector space models for improved bug localization," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 171–180.

[45] J. Sohn, Y. Kamei, S. McIntosh, and S. Yoo, "Leveraging fault localisation to enhance defect prediction," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2021, pp. 284–294.

[46] S. Haiduc and A. Marcus, "On the effect of the query in ir-based concept location," in *2011 IEEE 19th International Conference on Program Comprehension*, 2011, pp. 234–237.

[47] S. Haiduc, "Automatically detecting the quality of the query and its implications in ir-based concept location," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011, pp. 637–640.

[48] C. Mills, E. Parra, J. Pantiuchina, G. Bavota, and S. Haiduc, "On the relationship between bug reports and queries for text retrieval-

based bug localization," *Empirical Software Engineering*, vol. 25, 09 2020.

[49] J. M. Florez, O. Chaparro, C. Treude, and A. Marcus, "Combining query reduction and expansion for text-retrieval-based bug localization," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 166–176.

[50] S. Sinha and M. Harrold, "Analysis and testing of programs with exception handling constructs," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 849–871, 2000.

[51] (2022) Usage statistics of site elements for websites. [Online]. Available: https://w3techs.com/technologies/overview/site_element

[52] Oracle, "Jax-rs: Java api for restful web services," 2013, last accessed Jul. 2021. [Online]. Available: https://download.oracle.com/otn-pub/jcp/jaxrs-2_0_rev_A-mrel-eval-spec/jsr339-jaxrs-2.0-final-spec.pdf

[53] "Eager fetching is a code smell when using jpa and hibernate," https://vladmihalcea.com/eager-fetching-is-a-code-smell/, 2021, last accessed Aug. 2021.

[54] T.-H. Chen, S. W. Thomas, and A. E. Hassan, "A survey on the use of topic models when mining software repositories," *Empirical Software Engineering*, vol. 21, no. 5, pp. 1843–1919, 2016.

[55] G. Kul, D. T. A. Luong, T. Xie, V. Chandola, O. Kennedy, and S. Upadhyaya, "Similarity metrics for sql query clustering," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 12, pp. 2408–2420, Dec 2018.

[56] N. Arzamasova, M. Schäler, and K. Böhm, "Cleaning antipatterns in an sql query log," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 1751–1752.

[57] A. Islam and D. Inkpen, "Semantic text similarity using corpus-based word similarity and string similarity," *ACM Trans. Knowl. Discov. Data*, vol. 2, no. 2, Jul. 2008.

[58] N. Arzamasova, K. Böhm, B. Goldman, C. Saaler, and M. Schäler, "On the usefulness of sql-query-similarity measures to find user interests," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 10, pp. 1982–1999, Oct 2020.

[59] T.-H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 666–677.

[60] R. Singh, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Optimizing the performance-related configurations of object-relational mapping frameworks using a multi-objective genetic algorithm," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '16, 2016, pp. 309–320.

[61] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Diversity-based web test generation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 142–153.

[62] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, "Empirical comparison of black-box test case generation tools for restful apis," in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2021, pp. 226–236.

[63] G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid, "Class imbalance evolution and verification latency in just-in-time software defect prediction," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, pp. 666–676.

[64] T. Warszawski and P. Bailis, "Acidrain: Concurrency-related attacks on database-backed web applications," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 5–20.

[65] PetClinic. (2022) A sample spring-based application. [Online]. Available: https://github.com/spring-projects/spring-petclinic

[66] CloudStore. (2022). [Online]. Available: https://github.com/CloudScale-Project/CloudStore

[67] TPC-W. (2022). [Online]. Available: https://www.tpc.org/tpcw/

[68] BroadleafCommerce. (2022) Broadleafcommerce - enterprise ecommerce framework based on spring. [Online]. Available: https://github.com/BroadleafCommerce/BroadleafCommerce

[69] PublicCMS. (2022). [Online]. Available: https://github.com/sanluan/PublicCMS

[70] WallRide. (2022) Multilingual easy-to-customize open source cms made by java. [Online]. Available: https://github.com/tagbangers/wallride

[71] JeeWeb. (2022). [Online]. Available: https://github.com/white-cat/jeeweb

[72] bbs. (2022). [Online]. Available: https://github.com/diyhi/bbs

[73] A. S. Foundation, "Apache jmeter," 2021. [Online]. Available: https://jmeter.apache.org/

[74] E. Foundation, "The aspectj project," 2021. [Online]. Available: https://www.eclipse.org/aspectj/

[75] Oracle. (2020) Mysql server logs. [Online]. Available: https://dev.mysql.com/doc/refman/5.6/en/server-logs.html

[76] PostgreSQL. (2020) Error reporting and logging. [Online]. Available: https://www.postgresql.org/docs/13/runtime-config-logging.html

[77] Oracle. (2020) The general query log. [Online]. Available: https://dev.mysql.com/doc/refman/5.6/en/query-log.html

[78] "Spring framework," https://spring.io/, 2021, last accessed Aug. 2021.

[79] Oracle. (2020) The slow query log. [Online]. Available: https://dev.mysql.com/doc/refman/5.6/en/slow-query-log.html

[80] ——. (2020) Deadlocks in innodb. [Online]. Available: https://dev.mysql.com/doc/refman/5.6/en/innodb-deadlocks.html

[81] ——. (2020) An innodb deadlock example. [Online]. Available: https://dev.mysql.com/doc/refman/5.6/en/innodb-deadlock-example.html

[82] ——. (2015) Lockmodetype. [Online]. Available: https://docs.oracle.com/javaee/7/api/javax/persistence/LockModeType.html

[83] V. Mihalcea. (2019) How do lockmodetype.pessimistic_read and lockmodetype.pessimistic_write work in jpa and hibernate. [Online]. Available: https://vladmihalcea.com/hibernate-locking-patterns-how-do-pessimistic_read-and-pessimistic_write-work/

Wei Liu Wei Liu received a bachelor's degree in computer science and a master's degree in software engineering from Wuhan University of Technology, China, in 2012 and 2014, respectively. He is currently a Ph.D. student at the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. His research interests include software analysis, testing, and performance. Prior to it, he worked at Alibaba as a software engineer for two years.

Tse-Hsun (Peter) Chen Tse-Hsun (Peter) Chen is an Assistant Professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. He leads the Software PErformance, Analysis, and Reliability (SPEAR) Lab, which focuses on conducting research on performance engineering, program analysis, log analysis, production debugging, and mining software repositories. His work has been published in flagship conferences and journals such as ICSE, FSE, TSE, EMSE, and MSR. He serves regularly as a program committee member of international conferences in the field of software engineering, such as ASE, ICSE, ICSME, SANER, and ICPC, and he is a regular reviewer for software engineering journals such as EMSE and TSE. Dr. Chen obtained his BSc from the University of British Columbia, and MSc and PhD from Queen's University. Besides his academic career, Dr. Chen also worked as a software performance engineer at BlackBerry for over four years. Early tools developed by Dr. Chen were integrated into industrial practice for ensuring the quality of large-scale enterprise systems. More information at: http://petertsehsun.github.io/.