# WeBridge: Synthesizing Stored Procedures for Large-Scale Real-World Web Applications

GANSEN HU, Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, China
ZHAOGUO WANG*, Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, China
CHUZHE TANG, Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, China
JIAHUAN SHEN, Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, China
ZHIYUAN DONG, Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, China
SHENG YAO, Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, China
HAIBO CHEN, Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, China

Modern web applications use databases to store their data. When processing user requests, these applications retrieve and store data in the database server, which incurs network round trips. These round trips significantly increase the application's latency. Previous approaches have attempted to reduce these round trips by prefetching query results or batching database accesses. However, neither method can efficiently reduce the latency when some queries depend on previous queries' results. In real-world applications, nearly 50% of the queries depend on the result of other queries.

This paper presents WeBridge, the first system capable of synthesizing stored procedures for large-scale real-world web applications. First, WeBridge employs *concolic execution* technique to analyze the applications and generate stored procedures for *hot program paths*. Then, it seamlessly integrates the stored procedures into the application by extending the database access library. Finally, it improves the efficiency of the stored procedures with speculative execution. Evaluation using real-world web applications and workloads show that WeBridge achieves up to 79.8% median latency reduction and up to 2× peak throughput.

CCS Concepts: • **Information systems → Database transaction processing**; *Query optimization*.

Additional Key Words and Phrases: database-backed web applications, ORM, performance optimization, stored procedures, program analysis, concolic execution

*Corresponding author (zhaoguowang@sjtu.edu.cn)

Authors' addresses: Gansen Hu, Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, Shanghai, China, hugansen@sjtu.edu.cn; Zhaoguo Wang, Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, Shanghai, China, zhaoguowang@sjtu.edu.cn; Chuzhe Tang, Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, Shanghai, China, t.chuzhe@sjtu.edu.cn; Jiahuan Shen, Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, Shanghai, China, shenjiahuan@sjtu.edu.cn; Zhiyuan Dong, Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, Shanghai, China, zydong829@sjtu.edu.cn; Sheng Yao, Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, Shanghai, China, yao.sheng@outlook.com; Haibo Chen, Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, Shanghai, China, haibochen@sjtu.edu.cn.


ACM 2836-6573/2024/2-ART64
https://doi.org/10.1145/3639319

**ACM Reference Format:**
Gansen Hu, Zhaoguo Wang, Chuzhe Tang, Jiahuan Shen, Zhiyuan Dong, Sheng Yao, and Haibo Chen. 2024. WeBridge: Synthesizing Stored Procedures for Large-Scale Real-World Web Applications. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 64 (February 2024), 29 pages. https://doi.org/10.1145/3639319

## 1 Introduction

Latency is critical to web applications. Studies have shown that a 500 ms increase in latency can lead to a 20% decrease in overall traffic [75], and nearly half of the users will abandon a website if the page load time exceeds 3 seconds [106]. One primary reason behind the high latency is frequent communication between the web server and the database server [39, 40, 107–109]. As most web applications store their data in a database server, each data access incurs a network round trip. In a real-world e-commerce web application [7], when a user adds a product to the shopping cart, the application might necessitate over a hundred network round trips to retrieve and store data in the database server.

Two lines of research have been proposed to mitigate the database communication latency. The first is prefetching [36, 78, 90, 92], which involves issuing queries to the database as soon as the query parameters are available. Doing so asynchronously allows for the overlapping of database communication and application computation to hide latency. The second is batching [39, 40], which defers query execution until the query results become essential. This deferred execution technique enables the accumulation of multiple queries on the application side, thereby creating opportunities for batching. By issuing these deferred queries together, the number of network round trips and overall application latency can be reduced. However, these approaches have a fundamental limitation when it comes to optimizing the latency associated with queries that have dependencies on other queries. The limitation occurs because the computation of query dependencies solely takes place on the application side, necessitating extra round trips to retrieve the required data for computation.

This paper describes a new approach that reduces the number of network round trips incurred by issuing SQL queries even when they have dependencies. The approach leverages stored procedures [6] to execute dependent queries on the database side, eliminating the need for communication with the application. Our goal is to automatically synthesize stored procedures that benefit web applications. To achieve this, we must analyze web application APIs to understand what and how SQL statements are interactively issued. Prior works have applied static analysis methods on simple example web applications [37, 38, 92]. However, the direct application of static analysis to present-day web applications poses challenges. Many applications heavily rely on third-party libraries like object-relational mapping (ORM) layers and web frameworks. These libraries extensively employ *dynamic language features*, such as reflection in Java, to avoid code duplication. The utilization of dynamic features makes it exceedingly difficult to statically analyze applications whose behaviors are determined at run time [74]. Moreover, the widespread usage of these libraries often results in large-scale codebases in today's web applications, further complicating the existing static analysis approach [24].

We built WeBridge, the first system capable of synthesizing stored procedures for large-scale real-world web applications. The system adopts a dynamic program analysis technique— concolic execution [23, 32, 65, 87, 97, 98]—to analyze the database accesses. Using concolic execution, we re-executed *hot program paths*, guided by the observation that most requests in real-world workloads exercise only a few paths [70]. The re-execution allows us to collect SQL statements issued to the database and extract their control and data dependency information. With the extracted information, WeBridge constructs the stored procedures for hot program paths accordingly. Then, WeBridge seamlessly integrates the synthesized stored procedures into the application by intercepting all

**Original code fragment**

```
1:  @RequestMapping("AddCart/{uid}/{pid}")
2:  Response addCart(int uid, int pid) {
3:   // Get the user, cart, product tuples from the database
4:   User user = em.find(User.class, uid);                    [Q1]
5:   Cart cart = em.find(Cart.class, user.getCartId());       [Q2]
6:   Product product = em.find(Product.class, pid);           [Q3]
7:   // Check if the product is available
8:   if (product.getStock() > 0) {
9:    CartItem item = new CartItem(pid);
10:   cart.addItem(item);
11:   // Add item to the database tables
12:   em.save(Item.class, item);                             [Q4,Q5]
13:   product.stock = product.stock - 1;
14:   // Update product in the database
15:   em.save(Product.class, product);                       [Q6]
16:   return new Response("Succeeded");
17:  } else {
18:   // Log the "No stock" error to the LOG table.
19:   em.save(Log.class, new Log(pid, "NO_STOCK"));          [Q7]
20:   return new Response("No stock"); }}
```

Fig. 1. A simplified application code fragment abridged from Broadleaf's `addCart` API [7]. This API adds a product to a user's shopping cart. When the `product` is available (line 8), a single ORM invocation (line 12) issues two `INSERT`s to the `ITEM` table (Q4 and Q5) to add the `product` to the `cart`.

SQL statements and invoking stored procedures when appropriate. This integration method is transparent to the application, as it does not necessitate any changes to the application's source code. Additionally, WeBridge employs speculative execution to enhance the efficiency of the synthesized stored procedures. Finally, if the analyzed execution paths were encountered again, WeBridge invokes the stored procedures for better performance. Conversely, if the execution paths are never met before, WeBridge would revert to issuing the original application's interactive SQL statements.

We evaluated WeBridge with six real-world applications from GitHub, covering four different categories: e-commerce, blogging, forum, and configuration management. WeBridge has successfully synthesized stored procedures that consolidate interactive SQL statements for all applications. The results showed that, by serving hot paths with synthesized stored procedures, WeBridge achieves up to 79.8% median request latency reduction and 2× increase in peak throughput. To summarize, this paper makes the following contributions.

- Recognition of the significance and challenges of synthesizing stored procedures for real-world web applications.
- Design and implementation of WeBridge, the first system for synthesizing stored procedures for real-world web applications.
- Comprehensive evaluation using real-world web applications and workloads that validates the effectiveness of WeBridge.

## 2 Motivation and Challenges

### 2.1 Latency in Web Applications

Today, application servers and database systems are typically hosted on separate nodes, possibly in different data centers [35, 103]. Consequently, each database access incurs a round trip between the web server and the database server. Figure 1 shows a simplified code fragment from Broadleaf's `addCart` API [7] and Figure 2 shows the SQL statements issued by the ORM framework. There are either four or six network round trips, depending on the branching decision at line 8. We evaluated

**Derived stored procedure**

```
1:  CREATE PROCEDURE sp(uid_in INT, pid_in INT) BEGIN
2:   SELECT * INTO @userId_user, @cartId_user, ...          [Q1]
         FROM USER WHERE id = @uid_in
3:   SELECT * INTO @cartId_cart, ... FROM CART             [Q2]
         JOIN CART_ITEMS JOIN ITEM ON …
         WHERE id = @cartId_user
4:   SELECT * INTO @productId, @stock, ...                 [Q3]
         FROM PRODUCT WHERE id = @pid_in
5:   IF @stock > 0 THEN
6:    INSERT INTO ITEM(itemId, productId) VALUES (...)     [Q4]
7:    INSERT INTO CART_ITEMS(cartId, itemId) VALUES (...)  [Q5]
8:    UPDATE PRODUCT SET stock=stock-1 WHERE id=@productId [Q6]
9:    RETURN true
10:  ELSE
11:   INSERT INTO LOG(@productId, @eventName)             [Q7]
        VALUES (@productId, "NO_STOCK")
12:   RETURN false
13:  END IF END
```

Fig. 2. The derived stored procedure for Figure 1.

this example with separately deployed client, web server, and database server (within the same data center), simulating a realistic workload derived from [67]. We found that 56% of the request processing time is spent on network communication between the web server and database.

**Existing Solutions and Limitation.** Two approaches have been proposed to mitigate this latency. The first is prefetching [36, 78, 90, 92], which eagerly fetches query results as soon as a query's parameters are ready. This approach parallels computation and network to reduce the latency impact caused by round trips. In Figure 1, the parameters of $Q_1$ (line 4) and $Q_3$ (line 6) are ready at the beginning of the method. Thus, the application could prefetch these two queries asynchronously, in parallel with subsequent computations. The second approach is batching [39, 40], which defers the execution of SQL statements until the query result is required externally. In Figure 1, the results of $Q_2$ (line 5) and $Q_3$ (line 6) are not needed until line 8, so they can be lazily batched into one round trip at line 8.

However, both approaches fail to minimize the latency of queries if one control- or data-depends on the other. Unfortunately, such dependencies are common in web applications. In Figure 1, $Q_6$ (line 15) control-depends on $Q_3$ (line 6) since $Q_3$'s result determines whether $Q_6$ should be executed (line 8). $Q_2$ (line 5) data-depends on $Q_1$ (line 4) since $Q_2$'s parameter is derived from $Q_1$'s result. It is not feasible to prefetch SQL queries that depend on other queries in one round trip. This is because the execution decision or the parameter selection of the dependent queries can only be made after the execution of the preceding queries. For $Q_2$ in the example, only after $Q_1$ gets executed can $Q_2$ know what parameters it should use. As a result, $Q_2$ cannot get prefetched at the beginning of the API. Similarly, batching is not viable when there are dependencies among SQL queries. To see the impact of queries with dependencies, we analyzed the queries in Broadleaf and found that 46% of the SQL statements have control or data dependency on other queries.

**Opportunities: Stored Procedures.** Stored procedures are fully fledged database programs capable of expressing both data and control dependencies among SQL statements, providing opportunities to execute business logics without communication with the application. To showcase the effectiveness of stored procedures, Figure 2 shows a stored procedure that encapsulates all data accesses in Figure 1. The necessary round trips are reduced from six (four) to one, as shown in Figure 3. As a result, the end-to-end latency decreased by 59%, a significant improvement over existing solutions. Meanwhile, stored procedures can avoid repeatedly parsing and planning the same set of parameterized SQL statements. As a result, more processing power can be spent on

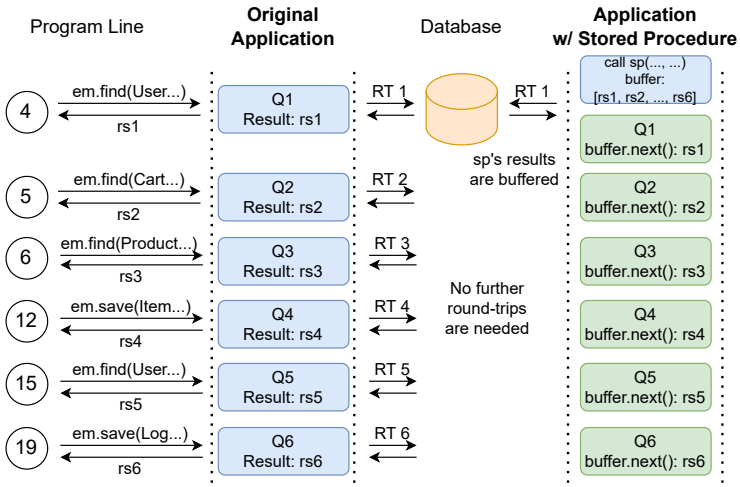


Fig. 3. Round trip (RT) comparison between the original application and one using a single stored procedure.

executing core business logics, which improves application throughput. For the above example, using stored procedure achieves 1.38× higher throughput.

Despite these benefits, stored procedures are still rarely used in web applications. To gain insights into the practical usage of stored procedures, we conducted an investigation on the 50 most popular open-source web applications on GitHub. By manual examination of source code and documentation, we found that none of them uses stored procedures. A recent report [88] also confirms this finding, showing that more than half of the production database workloads seldom, if ever, use stored procedures. One primary reason might be that stored procedure is notorious for the difficulties in development [57]. On the one hand, developers have to both manually write declarative data access and modification statements on their own, which is a challenging task that directly leads to the prevalence of ORM frameworks today, but also write control logics in a "low-level" and "arcane" procedural language [38].

Therefore, we ask: *Can we automatically synthesize stored procedures from large-scale real-world web applications today?*

## 2.2 Challenges

It is non-trivial to automatically synthesize stored procedures from real-world web applications. To the best of our knowledge, no existing work can achieve this goal. An intuitive way is using static program analysis techniques to obtain dependencies among database accesses, and construct mapping rules between ORM interfaces with SQL statements. Then, we could synthesize stored procedures with the dependency information in a rule-based manner.

Existing works [37, 38, 92] have already applied static analysis to web application source code. They use data flow analysis [22] and program dependence graph analysis [51] techniques to extract the control and data dependencies among database accesses, and construct dependency graphs for further optimizations such as prefetching [92], partitioning the execution [38], or code rewriting [37]. Unfortunately, real-world web applications heavily rely on language-level dynamic capabilities, which makes static analysis challenging [24, 74]. These dynamic capabilities allow applications to decide when and what SQL statements will be issued at runtime. For example, Java provides `ClassLoader`, a native utility to incorporate class definitions (i.e., data members and methods) at runtime [82]. Hibernate [61] uses this feature to achieve the lazy loading of database rows. In Figure 1, the loading of stock information of a product could be deferred until `product.getStock()`

is called at line 8. Internally, when invoking `em.find(Product.class, pid)` at line 6, this method internally creates a subclass of `Product` that overwrites the original `getStock()` method. The corresponding method in the subclass will issue appropriate `SELECT` statement(s) if the stock information has not been loaded from the database yet. Since the subclass definition is generated on the fly at runtime, it is non-trivial for static analysis methods to know the program behavior (e.g., what and when each SQL statement will be issued) at compile time. In addition to lazy loading, web applications also use language dynamism for aspect-oriented programming [100] and declarative transaction management [101], which potentially affect the behaviors of almost all their APIs and effectively render static analysis unfit for today's web applications.

Furthermore, constructing mapping rules between ORM interfaces and SQL statements is a daunting task. On the one hand, existing ORM frameworks expose numerous public interfaces actively used by web applications today. For example, we found hundreds of distinct ORM interfaces [10, 21, 47, 62, 102] used by the 50 most starred applications mentioned in §2.1. Meanwhile, even the seemingly simple `em.find()` interface in Figure 1 has rather complex internal logics—depending on the data class definition, it might generate complex equi-joins [104] and additional validation statements [25, 59]. Thus, it is non-trivial to determine the SQL statements issued by ORM-backed applications. Even worse, the large scale of codebases in real-world web applications makes static analysis techniques even more difficult. For example, the application in Figure 1 has 328k lines of code (LOCs) for its business logic implementation and the extensive use of third-party libraries, notably ORMs, bloats the code base size to 1.2M LOCs. In addition, we instrumented the JVM runtime to count the Java bytecode instructions and found that critical APIs like `addCart` [29] and `purchase` [30] execute around 840M and 710M bytecode instructions, respectively. This gives us some clues on why previous work [37, 38, 92] only evaluates TPC-C and TPC-W benchmarks or outdated small-scale applications like RUBiS or RUBBoS [83, 84].

## 3 WeBridge Overview

WeBridge is the first system that brings the benefits of stored procedures to real-world web applications without manual efforts.

**Insights.** WeBridge is based on two insights. First, advances in *concolic execution* [23, 32, 63, 65, 79, 87, 93, 97, 98] provide suitable tools to analyze programs that employ language-level dynamic capabilities and generate code at runtime. Unlike other dynamic analysis methods, such as taint analysis [95], concolic execution yields not only symbolic relationships between program variables, but also precise control dependencies among program statements for specific program paths. Such information is essential for maximizing the number of interactive SQL statements in the synthesized stored procedures. Second, while web applications have thousands of possible program paths, in practice, only a few *hot program paths* are frequently executed. Therefore, instead of analyzing all possible program paths at runtime, we can optimize hot program paths only, which significantly improves performance while reducing analysis effort. This observation is supported by [70] and confirmed by an e-commerce application trace dataset [67]. The dataset contains a user request history with request types and parameters. When replaying the trace on Broadleaf [7], an open-source e-commerce application, we found that only 12 distinct program paths were executed, with the two hottest paths accounting for 96.3% of the requests.

**Our Approach.** Based on the above insights, WeBridge transforms database accesses into stored procedure for only hot paths with concolic execution. To avoid runtime overhead, WeBridge offloads the analysis to an offline component. Figure 4 shows the architecture, consisting of an *offline compiler* and a *runtime library*. The *offline compiler* synthesizes stored procedures using application source code and recorded program paths as input. The *runtime library*, before synthesis,

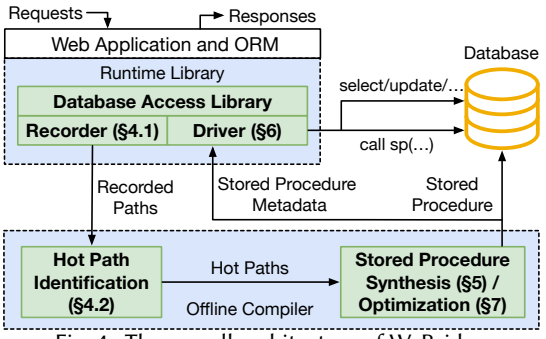


Fig. 4. The overall architecture of WeBridge.

records application execution paths and, after synthesis, intercepts interactive SQL statements with invocations to the synthesized stored procedures.

To elaborate, the recorder in the *runtime library* records the states of API execution, including user requests, execution results of SQL statements, and external method calls (§4.1). This information concisely represents the program paths and is sent to the *offline compiler*. Note that the *runtime library* is integrated at the database access library (e.g., JDBC [2]) level, which is transparent to both web applications and web frameworks, including ORMs. After collecting sufficient recorded paths, the compiler wakes up to synthesize stored procedures. It first identifies hot paths from recorded paths (§4.2), then concolically replays hot paths to produce a dependency graph (§5.1), and finally compiles the dependency graph into stored procedures (§5.2). Moreover, WeBridge optimizes the stored procedures code by reducing the number of instructions via speculative execution (§7). The generated stored procedures are then installed in the database and registered in the driver of the *runtime library*. At runtime, when executing the APIs that have synthesized stored procedures, the driver intercepts all query invocations. It instead invokes the stored procedure when the corresponding API tries to issue the first SQL statement, and buffers the result. With buffered results, WeBridge continues to execute the application's code and uses the buffered results to answer interactive SQL statements (§6.1). When the application tries to execute a query not captured by the stored procedure (aka a cold-path SQL), the driver resumes normal execution and issues interactive SQL statements.

Take the API in Figure 1 as an example. Each time this API is invoked, WeBridge records API inputs (e.g., the value of `uid`) and the results of SQL statements. It then replays the execution offline with the recorded information. Assuming the product-available path, i.e., one where the condition `product.getStock()>0` at line 8 is true, is frequently executed, WeBridge synthesizes a corresponding stored procedure for this path. Figure 2 shows the synthesized stored procedure, excluding lines 10–12, which correspond to another cold path. The stored procedure also returns markers indicating executed SQL statements (§6.2). Note that the condition which checks product availability is preserved in the stored procedure at line 5. After the stored procedure is installed, it will be invoked to process future requests. If a future hot-path request arrives, after invoking the stored procedure and buffering its results, the driver in the *runtime library* will find that all the interactive SQL queries made by this execution exactly match those performed by the stored procedure. Therefore, the driver will not issue any interactive SQL statements and use buffered results instead. When processing cold-path requests, i.e., the one whose execution finds the product is out of stock, the stored procedure only executes `Q1-Q3`, skipping `Q4-Q6` due to the branch condition at line 5. WeBridge driver can detect that only `Q1-Q3` have been executed and `Q7`, which sits on the cold path, is skipped. Therefore, it only uses the buffered results for the first three

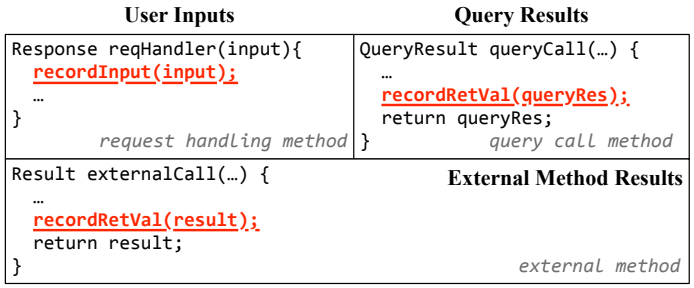


Fig. 5. WeBridge instruments the application to record the external states used in application.

queries at lines 4–6 in Figure 1 and issues the original SQL statement at line 19, where the driver interception ends.

## 4 Recording and Identifying Hot Paths

WeBridge records execution paths at run time during normal execution and identifies hot paths via offline replay. A hot path is one that has been replayed more times than a certain threshold.

### 4.1 Recording Paths

To record a path and further replay it deterministically, conventional solutions [43, 46, 48] record all the program inputs and track all shared memory accesses. However, the runtime overhead incurred by tracking shared memory accesses is significantly high, resulting in a latency increase of up to 100× [43, 54, 80, 86, 94]. Fortunately, we observe that modern web applications often intentionally avoid shared memory access. A survey [5] shows that most web services are architected in the *REST style* [52], which is a widely accepted architectural style that defines a set of unified APIs for client-server communication. The REST style requires that client requests include all the necessary information for processing, and *disallows any use of global application states* on the web server [69]. WeBridge assumes REST-style web applications, where there is no global states at the application side. Due to the absence of shared memory access, it suffices to only record external inputs of a request for deterministic replay.

To achieve this, WeBridge has devised a lightweight recording method that only records the external states, i.e., *user inputs*, *query results*, and *return values of external method calls* at runtime for each path. WeBridge utilizes existing instrumentation tools [1] to record these three kinds of external states. Figure 5 illustrates the instrumentation process. To record user inputs, WeBridge intercepts the request handling method and stores the request input at the beginning of the method. To record query results, WeBridge intercepts the database accesses calls and records the results before returning from the call. To record external method calls, such as I/O operations and retrieving current system time, WeBridge also intercepts invocations to these methods and records their return values.

### 4.2 Identifying Hot Paths

A path is hot if it is replayed over a predefined number of times. WeBridge replays each recorded path with external states on a separate application instance. This involves intercepting database access and external method calls, using recorded values to simulate the execution. During the replay, WeBridge collects the branching decisions of each individual branch instruction, which uniquely identifies a path. Formally, a path $P$ is defined as $P = [b_1, b_2, ..., b_n]$, which is a sequence of $n$ boolean branching decisions $b_i$. If $b_i = 1$, it signifies the $i$-th branch decision took the '*then*' branch; otherwise, the '*else*' branch was taken. Two recorded paths are considered the same if their sequences of branching decisions are identical.

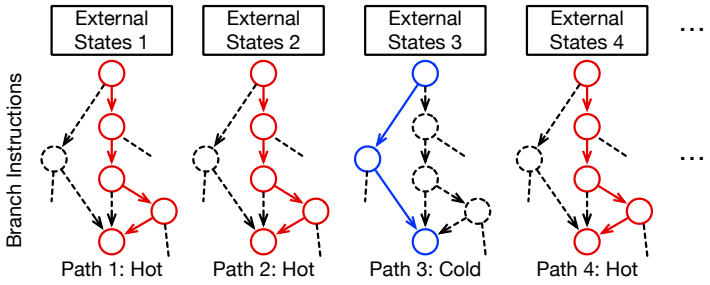


Fig. 6. An example on how to identify a hot path.

Figure 6 illustrates the process of identifying a hot path, with four recorded paths (path 1–4) along with their corresponding inputs denoted as external states 1–4. WeBridge replays each path and produces a sequence of recorded branch decisions for each path. For instance, path 1 records [0, 1, 0, 1] as branch decisions. Next, WeBridge compares the branch decisions with those of replayed paths'. For example, after replaying path 4, it compares the branch decisions with paths 1–3 and finds that paths 1, 2, and 4 share the same set of branch decisions (i.e., [0, 1, 0, 1]). This indicates that the same path has already been replayed twice (via states 1 and 2). Assuming the threshold of identifying a hot path is two, WeBridge regards the path replayed by state 4 as a hot path.

## 5 Synthesizing Stored Procedures

WeBridge synthesizes stored procedures for hot paths in two steps. It first replays the hot path and collects all the executed SQL statements. At the same time, it uses concolic execution to extract the control and data dependency of SQL statements. The extracted results are then represented as a dependency graph. Second, it synthesizes stored procedures from the dependency graph with a rule-based rewrite. Before discussing the details, we briefly introduce necessary background on concolic execution.

**Concolic Execution.** Concolic execution [65, 98] is a technique that captures the precise data and control dependencies of concrete program executions. Given a program and a concrete input, a concolic execution engine interprets the program both concretely and symbolically. For example, given a concrete input variable `a` with a value of 10, interpreting a single-line program `b=a+1` assigns the value 11 to the program variable `b` and creates a symbolic variable `b_sym` with the expression `ADD(a_sym,1)`. Two additional data structures are used to capture the precise dependencies: a *symbolic store* and a *path condition*. The symbolic store maps program variables to their corresponding symbolic variables. Each symbolic variable contains a symbolic expression that represents how it is computed based on other symbolic variables. The symbolic store is read and updated each time the execution engine interprets an assignment instruction. The path condition is a special Boolean symbolic expression—the conjunction of all branch conditions taken along the current executing path. It is updated each time the execution engine enters a new basic block (i.e., when a branch is taken). Therefore, it predicates whether certain program inputs would cause the program to make the same branching decisions as the current concrete execution.

### 5.1 Constructing Dependency Graphs

A dependency graph is a directed graph, where each vertex represents the issue of a SQL statement in the path during execution. Each edge represents that the SQL statement in the source vertex is issued before the SQL statement in the target vertex. The detailed definition of the vertex is shown below.

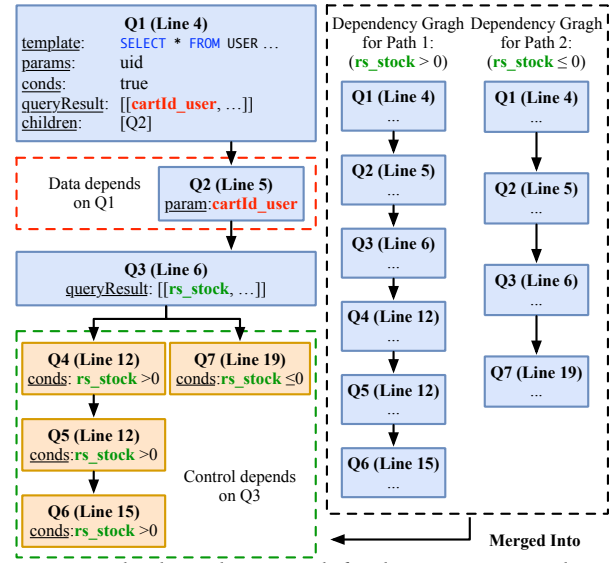


Fig. 7. The dependency graph for the Figure 1 example.

```
1: Node {
2:   String           template;     // sql statement template
3:   Expression[]     parameters;   // parameters expression
4:   Constraint[]     conditions;   // control flow constraints
5:   Expression[][]   queryResult;  // query result of the sql
6:   Node[]           children;  }  // successors of the node
```

The vertex consists of five parts: `template` and `parameters` represent the query template string and parameters, where each parameter can be a constant value or a symbolic expression. WeBridge currently supports `SELECT [FOR UPDATE]`, `SELECT INTO`, `UPDATE`, `DELETE`, `INSERT`, `BEGIN`, `COMMIT`, and `SET VARIABLE` statements. `conditions` represent the set of path conditions leading to this query statement. `queryResult` is the execution result, which is a two-dimensional array with potentially multiple rows and columns. `children` links a vertex with other vertices in the dependency graph, indicating the SQL statements executed after the vertex.

To illustrate, the left half of Figure 7 shows the dependency graph extracted from the application code of Figure 1. It contains seven vertices, each corresponding to a SQL statement. For instance, `Q1` (line 4) in Figure 1 is the root vertex in the graph, with the query string "`SELECT * FROM USER WHERE id = ?`" as the `template`. The `conditions` are `true` since `Q1` is unconditionally executed. The input `uid` from user requests serves as the `param` in `Q1`. `queryResult` of `Q1` is the selected user information, where `cartId_user` serves as the `param` of `Q2`, indicating `Q2` data depends on `Q1`.

WeBridge constructs a dependency graph for a hot path in three steps. Algorithm 1 outlines the dependency graph construction process. First, it performs concolic execution on the hot path to track dependencies among queries and external states. Specifically, WeBridge re-executes the identified hot path, marking all external states as symbolic values to track the dependency among queries and external states. In this way, all program computations related to external states are tracked in the symbolic store and path condition. Second, WeBridge intercepts database access API calls to create vertices and edges accordingly. When a query is issued via such API, WeBridge extracts the query's template, parameter expressions, and current path conditions to construct a vertex in the dependency graph. WeBridge then links the constructed vertex to the appropriate position of the dependency graph. Finally, WeBridge merges dependency graphs constructed from

**Algorithm 1:** Dependency graph construction.

```
1  ConstructDG(ExternalState state, Instruction[] instructions):
2  vertex = nil; curConds = true; graph = emptyGraph; store = {}
3  markSymbolic(state.inputs, store) // mark the input state as symbolic
4  foreach instr in instructions:
5    switch instr:
6      case instr is 'invoke prepareQuery':
7        vertex = createVertex(instr.sql)
8      case instr is 'invoke setParam':
9        vertex.parameters.add(instr.idx, store[instr.param])
10     case instr is 'invoke execute':
11       vertex.conditions = curConds
12       queryRes = nextQueryResult(state.queryResults)
13       // continue execution with symbolic query result
14       markSymbolic(queryRes, store)
15       vertex.queryResult = store[queryRes]
16       // append the new vertex to the tail of the graph
17       graph.tail.children.add(vertex); graph.tail = vertex
18   // execute the instruction concolically
19   ConcolicExecute(instr, curConds, store)
20   if instr is 'external call':
21     markSymbolic(nextReturnedValue(state.returnValues), store)
22 return graph
23
24 ConcolicExecute(Instruction instr, PathCondition pc, SymbStore s):
25 if instr is 'if e then s_true else s_false':
26   if e == true: pc = pc ∧ store[e]; goto 's_true';
27   else          pc = pc ∧ ¬store[e]; goto 's_false';
28 if instr is 'assign x=e': store[x] = e
29 if instr is 'binOp x y to z':
30   // creating a binary symbolic expression, and updates the symbolic store
31   store[z] = binOpExp(store[x], store[y], instr.op)
32 ...
33
34 MergeGraph(Graph g, Graph t):
35 // g: existing dependency graph, t: new graph constructed from another hot path
36 equalVertices = longestEqualVertexSequence(g, t)
37 foreach vertex in t:
38   if vertex in equalVertices:
39     n = g.getEqualityVertex(vertex)
40     n.conditions = n.conditions ∨ vertex.conditions
41     // adjust the parent and child vertices for the merged vertex n accordingly
42     n.children.add(vertex.children)
43 return g
```

each new hot path into the existing dependency graph. The final merged graph retains the query and dependency information of all hot paths.

**Concolically Executing Hot Paths.** This part involves (i) marking the external state as symbolic (lines 3, 14, 21) by associating each variable in *state* with an initial symbolic value in *store*, then (ii) updating the path condition *curConds* and the symbolic *store* based on the executed instruction (line 19). Line 24–31 shows how *curConds* and *store* get updated for branching, assignment, and

binary operation instructions. For instance, consider a binary instruction 'add x y to z' that adds the value of x and x to variable z. During concolic execution, a new symbolic expression binOpExp(sym_x,sym_y,add) representing the addition of symbolic variables sym_x,sym_y is associated to variable z. Note that we only include a part of the instructions for ease of presentation, and refer readers to [26, 31, 32, 87, 97, 98] for other types of instructions in various languages.

**Creating Dependency Graphs.** The vertices and edges in the dependency graph are created when the database access APIs are invoked. To achieve this, WeBridge intercepts each invocation instruction to these APIs, then parses the SQL query templates and parameters from the instruction operands. WeBridge assumes the application uses three database access APIs:

- prepareQuery(String sql): Initialize a query with the template string sql. WeBridge extracts the SQL template string from the instruction's operand and creates a vertex (lines 6–7).
- setParam(int idx, Object param): Set the *idx*-th parameter of the SQL statement to *param*. WeBridge extracts the symbolic representation of the parameters from the symbolic store, and add them to the vertex (lines 8–9).
- execute(): Issue the SQL query statement to the database for execution. WeBridge first records the path condition—encoding the control flow constraints leading to the execution of the query—in the vertex (line 11). Next, the symbolic representation of the query execution result is added to the vertex (line 15), where a vertex completes its creation. Finally, the vertex in linked to the tail of the dependency graph (line 17).

Notice that the dependency graph construction algorithm is language agnostic and only requires the database access libraries (e.g., JDBC) to provide the functionalities of the above APIs. Algorithm 1 ends when all instructions along the path finish re-execution.

**Merging Dependency Graphs.** WeBridge combines two vertices from different dependency graphs into a single vertex if they are equal. Specifically, vertex a and vertex b are equal if a.template is the same with b.template, and each parameter in a.parameters is the same with each parameter in b.parameters. Merge begins by finding the longest sequence of *equal* vertices among two graphs (line 36). Then, each vertex in the new graph t, which is also present in the equalVertexes, is merged into the existing graph g. When merging two *equal* vertices, the path conditions are updated to the disjunction of the two vertices (line 40). The rest of the vertices that cannot be merged are simply linked to the children of the merged vertex n (lines 42).

As shown in Figure 7, the right half contains two dependency graphs constructed for path 1 (stock > 0) and path 2 (stock $\leq$ 0). The dependency graph for path 1 contains the query execution sequence from Q1 to Q6, while path 2 contains Q1-Q2-Q3-Q7. Thus, the longest sequence of vertices in the two graphs that have *Equality* property is Q1-Q2-Q3. These vertices are then merged into the leftmost graph. To illustrate, the vertex of Q1 has the path condition true in both path 1 and 2. Thus, the merged Q1 has the path condition true $\vee$ true, which is also true. For vertices that do not have the *Equality* property (i.e., Q4-Q5-Q6 and Q7), they are linked to the children vertices of Q3, with path conditions unchanged.

### 5.2 Generating Stored Procedures

Given a dependency graph, WeBridge generates stored procedures by transforming each vertex into stored procedure code in a rule-based manner. These rules turns the query template string, symbolic parameters, and path conditions into stored procedure code. Due to the limitation of our concolic execution engine and the database support for stored procedures, WeBridge needs to split the dependency graph upon inconvertible computations.

**Inconvertible Computations.** There are two types of computations that WeBridge currently does not convert: (i) external method calls, such as java.lang.System.currentTimeMillis(),

**Algorithm 2:** Stored procedure generation.

```
SplitGraph(Graph graph):
// cut the graph for each vertex depending on states that are inconvertible
graphSet = emptySet; startPoints = endPoints = emptyList
// 1. identify the starting points
foreach vertex in graph:
  if vertex contains computation that cannot express in stored procedure, or an external call
  is executed between two vertices:
    startPoints.add(vertex)
// 2. identify the ending points
foreach vertex in graph:
  if vertex contains no child vertices, or all child vertices are in startPoints:
    endPoints.add(vertex)
// 3. construct subgraphs
foreach vertex in startPoints:
  // construct subgraphs from starting to ending points
  subGraph = emptyGraph; subGraph.root = vertex
  foreach v between vertex and endPoints:
    subGraph.add(v)
  graphSet.add(subGraph)
return graphSet

SynthesizeSP(Graph subgraph):
SoredProcedure sp = emptySp
// iterate through each vertex in the subgraph
foreach vertex in subgraph:
  qCode = constructQueryCode(vertex.template,
vertex.parameters, vertex.queryResult)
  qBlock = constructIfThenCode(vertex.conditions, qCode)
  sp.addSourceCode(qBlock)
  // identify stored procedure parameters
  foreach expr in vertex.parameters and vertex.conditions:
    if expr depends on symbols not defined in graph:
      sp.addInputParameter(expr)
return storedProcedure
```

because their implementations are unavailable to WeBridge's concolic engine, and (ii) array operations, such as bytecode `iaload` and `iastore`, because MySQL, the database system that WeBridge currently builds on, does not provide corresponding array types and operations in its stored procedures. For other computations, we could always construct translation rules to express them in stored procedures, which we will elaborate on later. To support the first case, we could model the semantics of those calls and craft specific rules. To support the second case, we could utilize a more capable database system. These are left for future work. To track the dependencies among SQL statements and inconvertible computations, we extend Algorithm 1 to mark the result of all inconvertible computations as symbolic and use concolic execution to track such dependencies.

**Splitting Dependency Graphs.** To split a graph into subgraphs, WeBridge first identifies the starting and ending points for each subgraph. Then, it puts all vertices from the starting point to the ending point into the subgraph. A vertex is a starting point if it contains inconvertible computations in path conditions or parameters, or an external call is executed between two consecutive vertices. The ending points are vertices with no child node or only starting points as child nodes. With

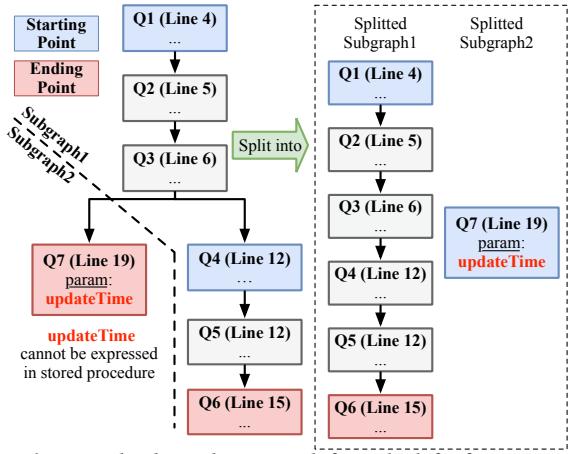


Fig. 8. Dependency graph splitting. The dependency graph from the left of Figure 7 is split into two subgraphs.

the identified points, WeBridge constructs subgraphs containing all the vertices from the starting point to the ending points. Algorithm 2 shows the graph-splitting process. It starts by identifying the set of `startPoints` and `endPoints` using the mentioned criteria (lines 5–11). Each starting point serves as the `root` vertex of a subgraph (line 15), and it adds all the vertices from the `root` to `endPoints` to the subgraph (lines 16–18).

To illustrate, assuming we add an additional parameter, `updateTime` to `Q7` (line 19) in Figure 1, which is derived from an external function call to get the current system time. Figure 8 demonstrates the corresponding dependency graph. As the stored procedure cannot express computation derived from an external function, the dependency graph on the left side in Figure 8 is split into two subgraphs that do not contain inconvertible computations. Each subgraph corresponds to a specific code path: subgraph 1 corresponds to the code that executes `Q1-Q6`, while subgraph 2 corresponds to the code that only executes `Q7`. Notice that we do not lose control or data dependency among queries after the split (e.g., we still know that `Q7` control depends on `Q3`). Such information is used to determine the execution order of stored procedures.

**Synthesizing Stored Procedures from Subgraphs.** Given a subgraph, WeBridge synthesizes the source code for a stored procedure in a rule-based manner, similar to existing source-to-source compilers [8, 9, 89]. These rules are manually crafted and assumed to preserve semantics. Specifically, WeBridge currently includes 71 transformation rules, which are sufficient to handle the six applications used in our evaluation. These rules encompass 21 arithmetic operations (e.g., `+/-/*/%`), 6 comparisons (e.g., `!=/==/>=`), 19 data mapping and type casting operations (e.g., casting `long` to `int`), and 25 string manipulations (e.g., `toUpper()`).

The `SynthesizeSP()` function in Algorithm 2 illustrates the transformation from a subgraph to a stored procedure. The stored procedure source code consists of two parts: the code block that issues SQL query statements (lines 25–28) and the necessary input parameters (lines 30–32). WeBridge synthesizes the code block in two steps. First, concatenate the SQL statement template string with appropriate parameters (line 25). Parameters in the vertex are represented in symbolic form, and we use transformation rules to rewrite them into equivalent stored procedure syntax expressions. For instance, `binOpExp(sym_x,sym_y,add)` is transformed into `x+y`. Additionally, each variable in `queryResult` is assigned to the execution result of the SQL statement, allowing subsequent queries to access results from previous queries. Second, wrap the above code block with a conditional statement (line 27): `IF(pc is TRUE) THEN (issue query)`, where `pc` is rewritten from the `conditions` of the vertex. In this form, the query executes only if `pc` evaluates to `true`.

```
1:  CREATE PROCEDURE sp1(uid_in INT, pid_in INT) BEGIN
2:    DECLARE userId_user INT; DECLARE cartId_user; DECLARE mk_1 TINYINT=0;
3:    SELECT * INTO @userId_user,@cartId_user FROM USER WHERE id=@uid_in  [Q1]
4:    SELECT @userId_user,@cartId_user;SET mk_1 = 1;
5:    SELECT * INTO … FROM CART JOIN … ON … WHERE id = @cartId_user       [Q2]
6:    SELECT * INTO … FROM PRODUCT WHERE id = @pid_in                     [Q3]
7:    IF @stock > 0 THEN
8:      INSERT INTO ITEM(itemId, productId) VALUES (…)                    [Q4]
9:      INSERT INTO CART_ITEMS(cartId, itemId) VALUES (…)                 [Q5]
10:     UPDATE PRODUCT SET stock=stock - 1 WHERE id = @pid_in             [Q6]
11:   END IF
12:   SELECT @mk_1, …; END
13: CREATE PROCEDURE sp2(stock INT, pid_in INT, time BIGINT) BEGIN
14:   IF @stock ≤ 0 THEN
15:     INSERT INTO LOG(pId, eventName, updateTime) VALUES (…, time)      [Q7]
16:   END IF END
```

Fig. 9. Synthesized stored procedures for Figure 8.

To synthesize the input parameters, WeBridge first identifies all symbolic expressions that depend on a vertex in other subgraphs, inputs, or external method return values (values not contained in the subgraph). Then, for each identified symbolic expression, synthesize a corresponding input parameter using the same name and type of the expression (line 32).

Figure 9 shows the synthesized stored procedures for the subgraphs in Figure 8. Subgraph 1 is transformed to `sp1` (upper half of Figure 9) and subgraph 2 to `sp2` (bottom half). Due to space limits, we omit the query result selection statement (line 4) for `Q2`–`Q7`. The usage of variable `mk_1` (lines 1,4,12) is explained in §6.

**Remarks on Preserving Transaction Boundaries.** Note that WeBridge places the SQL statements, including `BEGIN` and `COMMIT` commands, unmodified in the generated stored procedures. As a result, the generated stored procedure can contain multiple transactions or partially overlap with active transactions. Major database systems, such as MySQL and Oracle, treat stored procedures simply as batches of statements and they do not implicitly start or commit a transaction for the procedure. This means that if transactions start before or commit after the stored procedure, they will not be accidentally terminated early by the stored procedure. As a result, the generated procedure neither enlarges nor reduces transaction boundaries from the original application.

## 6 Integration into Web Applications

WeBridge driver automatically integrates the generated stored procedures into the running application, issuing stored procedures instead of the original interactive SQL statements for all database accesses. For unoptimized cold paths, the driver automatically switches back to issue interactive SQL statements.

### 6.1 Serving Hot Paths with Stored Procedures

When a hot path request arrives and the first SQL statement in the API is invoked, the driver intercepts the SQL statement and instead invokes stored procedures with corresponding parameters. These parameters are collected in the same way as described in §4.2, as they are external states. The driver then waits for the database to complete execution and return the results of all statements from the stored procedure. These results include query result sets and the execution status of writes, such as the number of rows affected. Finally, the driver buffers the results and resumes the execution of the application. For the invocations of the first and subsequent SQL statements, WeBridge answers them with corresponding buffered results from the buffer, without resorting

to the database. Each buffered result is used only once and is discarded after it has been used to answer one query.

To illustrate, assume WeBridge generated a stored procedure Figure 9 by a hot path collected in application Figure 1, where *stock>0* always holds. When the application invokes `Q1` (line 4 in Figure 1), the driver intercepts it and executes a stored procedure invocation statement—*call sp1(uid, pid)*—with collected concrete parameters of *uid* and *pid*. The driver then suspends the application's execution and waits for the database to complete the execution of the stored procedure. Once the database completes execution, the driver receives the execution result of `Q1-Q6` and buffers them in memory. It then resumes the application's execution and directly returns the buffered results for subsequent calls to `Q1-Q6`.

### 6.2 Fallback Mechanism for Cold Paths

WeBridge invokes a stored procedure upon the first SQL statement arrives, regardless of whether it is on a hot path or a cold path. When a cold path arrives, the stored procedure prevents the execution of any SQL statements that should only be executed on hot paths. This is because a path is determined solely by the external states (§4.2), which are all tracked by concolic execution (§5.1). The path conditions in the generated stored procedure will also reflect changes in external states that can lead to a cold path. For example, let's consider *sp1* in Figure 9, which is generated from a hot path. Assuming a cold path arrives where *stock>0* evaluates to false, the stored procedure tracks the derived expression *stock>0* as a condition to execute `Q4-Q6`, since *stock* comes from the result of `Q3`. If the *stock* number changes and result in a cold path, the stored procedure correctly stops issuing SQL statements at line 7, and returns the execution results of `Q1-Q3` to the driver in the application. The driver is aware that only a part of the SQL statements is executed and only returns buffered results to the following calls to `Q1-Q3`. After that, the driver switches back to invoke interactive SQL statements for the following SQL statements (i.e., `Q7` in the unoptimized cold path).

To perform this check, we associate each query in the stored procedure with a "marker" variable indicating whether it has been executed (line 2 in Figure 9). Markers are boolean variables individually declared and set for each query. For each query *i*, we add a "`SET marker_i=true;`" statement right after the query statement (line 4). All values of markers are returned to the driver by an additional "`SELECT @marker_1, @marker_2,...;`" statement added at the end of stored procedure (line 12). By analyzing returned markers, WeBridge knows which queries were not executed and switches back to normal execution accordingly.

### 6.3 Exception Handling

Exceptions are commonly used in web applications to report and handle runtime errors. WeBridge ensures that exceptions that would be thrown in the original application are thrown as is at the correct timing, and no new exceptions will be thrown by WeBridge. The only chance that WeBridge might alter the exception behavior is when it intercepts JDBC calls. Therefore, we thoroughly examined all potential exceptions that could be thrown by the MySQL JDBC driver, according to its specification [15]. Since stored procedures have no impact on the exceptions for statement-wise execution errors, if such exceptions occur during stored procedure execution, WeBridge only needs to catch and re-throw them when the application has reached the triggering statement. For stored procedure–specific exceptions, which only occur at the start of stored procedure invocation according to the specification, WeBridge will simply fall back to executing the original web application's code. This is safe because no statement has been executed yet, and the application will run as if WeBridge is absent.

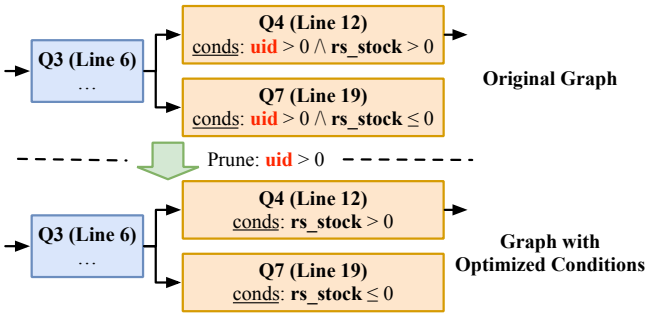


Fig. 10. An example of the path condition pruning. The constraint `uid≥0` of `Q4`, `Q7` is pruned, because their neighbor vertices also contain the constraint `uid≥0`.

## 7 The Speculative Execution Optimization

We introduce a speculative execution optimization to boost the performance of stored procedures. It is important to note that speculative execution is orthogonal to existing optimizations on procedural code [49, 53, 57, 68, 91].

**Pruning Unnecessary Path Conditions.** To avoid costly path condition evaluation inside stored procedures, WeBridge proposes a heuristic to speculatively remove path conditions that *solely* distinguish cold from hot paths. The heuristic is to eliminate constraints that are common among the vertex's *neighbor* vertices. Such constraints yield the same result in all paths and do not help distinguish different hot paths. WeBridge first identifies the neighbor vertices for each vertex in the dependency graph. Then, for each constraint of the vertex, WeBridge checks if it also exists in all other *neighbors*. If so, the constraint is pruned. If a vertex does not have any *neighbors*, its path condition can be simplified to "`true`". Figure 10 illustrates an example. `Q4` and `Q7` are neighbors of each other in the dependency graph. The constraint "`uid ≥ 0`" is common in both `Q4` and `Q7`, so that it can be removed.

**Handling Uncertain Side Effects.** Pruning path conditions can cause stored procedures to execute incorrect SQL statements. For read-only stored procedures, we could simply revert to normal execution in such situations. However, if stored procedures contain side effects (aka writes), WeBridge should only commit those effects when the speculation is correct. This is achieved by reusing the application transactions: when generating stored procedures, WeBridge now additionally splits at the end of each write transaction and excludes the final commit of such transaction from the generated stored procedures. In this way, after procedure execution, those write transactions remain active and WeBridge has a chance to validate their execution. Therefore, WeBridge can decide whether to commit or roll back. In addition, WeBridge adds *savepoints* [3, 4] before each write statement, so that it can undo only the incorrect writes while maintaining the result of previous correct statements. For standalone writes, although they are conceptually a "begin-write-commit" transaction, WeBridge instead treats them as inconvertible operations. Otherwise, splitting them into "begin-write" and "commit" causes additional round trips.

**Validation and Recovery.** To validate if the speculation is correct, WeBridge verifies whether the executed SQL statements in stored procedure match the ones requested by the application. It first checks the *marker* variables for each query (§6.2) to determine whether a query is executed. Then, for executed queries, WeBridge compares the SQL query to be issued by the application with the one executed in the stored procedure. If the query strings and parameters are not the same, WeBridge considers that as an incorrectly executed SQL query and recovers from it. If the first incorrect statement detected is a read, WeBridge rolls back the transaction to the nearest savepoint *after* that read, and discards the cached results of that and subsequent queries. If there is no such savepoint (i.e., no write exists after that read), then a rollback is unnecessary and WeBridge only

discards the cache accordingly. If the first incorrect statement detected is a write, WeBridge rolls back the transaction to the nearest savepoint *before* that write, and discards the cached results of subsequent queries. Note that there is always a corresponding savepoint for each write. Finally, for queries not executed, WeBridge falls back to normal execution.

**Remarks on Preserving Transaction Boundaries.** Note that WeBridge reuses the original application transactions to ensure correctness. It only excludes the last commit from the generated procedure, without removing or adding any read or write operation. Therefore, it preserves the original transaction boundaries.

## 8 Correctness

Due to space constraints, we only sketch a proof for the correctness of WeBridge. A full formal proof is available at [16].

**Assumptions.** WeBridge assumes REST [52] style web applications, where there is no application-side shared states. Consequently, the application code converted by WeBridge would only read/write the local states of each request handler and is thus deterministic. This means that, given the same local state and the same return values of both external method calls and SQL statements, a block of bytecodes would always transition the corresponding request handler's local state to the same next state. Nondeterministic computations are only allowed via external method calls, which is true for all the applications we evaluated. WeBridge also assumes that database states are only updated by SQL statements and SQL statements are also deterministic, which means that, given the same database state, a SQL statement would always output the same result and transition the database into the same next state. If a SQL statement contains nondeterministic function calls like `random()`, we could always factor them out as external method calls, and pass the return values as SQL parameters.

**Proof Sketch.** The correctness condition of WeBridge states that, for any execution possible with WeBridge, there is always an equivalent execution possible in the original application. We first discuss the case of sequential requests. In this case, the key is to prove that, if a SQL statement whose cached result is used by the API, then without WeBridge, the original API would issue the same SQL statement, given the same user input, external method call results, and prior SQL results. Applying this to the SQL statement sequence issued by the WeBridge-transformed API, we can prove that the database state is always transformed the same way with and without WeBridge. Similarly, we prove that local states are also transformed the same. As a result, any execution possible with WeBridge is also possible without WeBridge. When there are concurrent requests, we need to additionally consider how one API's SQL statement sequence issued inside a stored procedure might be interfered by another API's conflicting writes. We prove that, when such writes interleave with a stored procedure, the execution is equivalent to one under the original application with those writes interleaving corresponding source SQL statements in the same way.

## 9 Evaluation

We evaluate WeBridge to answer the following questions.

- Does WeBridge effectively improve the performance of real-world applications, both on latency and throughput?
- How does WeBridge compare with existing works that reduce the latency of web applications?
- How much performance overhead does WeBridge incur?

### 9.1 Experiment Setup

**Implementation.** Our approach is language-agnostic and applies to applications written in both statically typed and dynamically typed languages. Our current implementation, WeBridge, targets

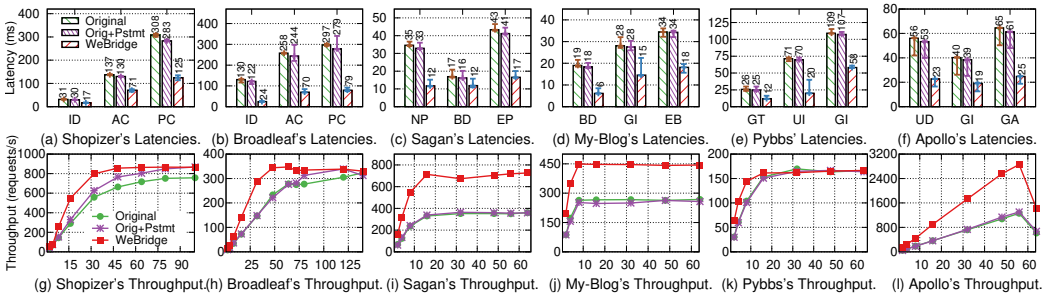


Fig. 11. End-to-end API latency (a–f) and throughput (g–l) results for the different APIs from the six applications evaluated. For latency results, X axis shows the APIs evaluated for each application, each rectangle shows the P90 latency, and the upper/lower error bars indicate P99/median latency. For throughput results, the workload requests each API with the same probability.

| Names | Category | Stars | LOCs | Commits | ORM Library |
|---|---|---|---|---|---|
| Broadleaf [7] | E-commerce | 1.6K | 328K | 18,086 | Hibernate |
| Shopizer [11] | E-commerce | 3.2K | 130K | 1,148 | Hibernate |
| Apollo [18] | Configuration Management | 28.4K | 59K | 2,849 | Hibernate |
| Sagan [20] | Blogging | 3.1K | 20K | 3,038 | Hibernate |
| My-Blog [13] | Blogging | 3.3K | 4.2K | 92 | MyBatis |
| Pybbs [12] | Forum | 1.7K | 15K | 931 | MyBatis |

Table 1. Details of the applications used in the evaluation.

Java applications. For the *runtime library*, we implemented a library that records the user input and external method calls via Java instrumentation API [64], and extended MySQL JDBC driver to record each query's `ResultSet`. For the *offline compiler*, we developed a concolic execution engine based upon Zero-Assembler [81] interpreter of OpenJDK 8, consisting of around 8k LoCs in C++. The stored procedure synthesizer is implemented as a separate Java library in about 4k LoC in C++ with 71 transformation rules. It produces MySQL-compatible stored procedures. The source code of our implementation is available at [16].

**Baselines.** We compare WeBridge-transformed applications with the following baselines. `Original`, which are the original applications without any modification. `Orig+Pstmt`, which are the original applications with database-side prepared statements enabled. Prepared statements allow the database to reuse the query parsing results, saving CPU time, a feature similar to stored procedures. `Prefetch+`, which uses the prefetching approach [78, 92] to hide the database round trips. The SQL statements and dependency information for `Prefetch+` are acquired via concolic execution rather than static analysis. `Sloth-Sim`, which uses the Sloth [39, 40] algorithm to reduce the database round trips via batching. Since we don't have access to Sloth's implementation, we report the theoretical round trip reduction instead.

**Applications.** We used six open-source Java real-world applications found on GitHub, as shown in Table 1. These web applications cover four different categories: e-commerce, blogging, forum, and configuration management. They are the most starred applications in each category on GitHub and have been under active development for 4–10 years. All have been adopted for real-world production use (e.g., Broadleaf has been used by 147 companies [14], and Apollo has been used by 290+ companies [17]). They use either Hibernate or MyBatis as the ORM library.

| | Broadleaf | | | Shopizer | | | Apollo | | |
|---|---|---|---|---|---|---|---|---|---|
| **API** | ID | AC | PC | ID | AC | PC | UD | GI | GA |
| **#HP** | 2 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 |
| **#SP** | 5 | 21 | 35 | 3 | 22 | 24 | 7 | 8 | 8 |
| **LOC** | 23 | 28 | 42 | 319 | 78 | 89 | 27 | 24 | 28 |
| **#IC** | 1 | 4 | 5 | 0 | 4 | 4 | 1 | 1 | 1 |
| | **Sagan** | | | **My-Blog** | | | **Pybbs** | | |
| **API** | NP | BD | EP | BD | GI | EB | GT | UI | GI |
| **#HP** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **#SP** | 3 | 1 | 5 | 1 | 1 | 2 | 2 | 6 | 2 |
| **LOC** | 72 | 561 | 51 | 139 | 42 | 118 | 32 | 43 | 829 |
| **#IC** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 2. Statistics of generated stored procedures. HP: hot paths, SP: stored procedures, LOC: lines of code, IC: inconvertible operations

**Workloads.** We made our best effort to build realistic workloads. We collected real-world application traces and workload statistics (e.g., the number of blog posts on a website), to build our experiment workloads for the applications in each category. For e-commerce applications, the workload is based on a real-world e-commerce user behavior dataset [67], with 285M user events over seven months from a large real-world e-commerce website. This dataset contains three types of user events: viewing a product, adding a product to a shopping cart, and purchasing a product. We populated the database tables with 302.2k users, 16.7k products, 62 categories, and 344 brands based on the metadata of the provided dataset. For blogging applications, the workload is based on a website running one of the applications (e.g., Sagan [20] has a running website *spring.io* [19]). Based on the statistics we collected from the website, we populated the database with 5k blogs and eight projects. For the forum application, we built the workload based on a website running an application similar to the application (e.g., we used statistics collected from Lobsters [60], which is also a forum application with similar functionalities, to Pybbs [12]). The database is populated with 9.2k users, 40k stories, and 120k comments. For the configuration management application, since most companies use it internally, we did not find publicly available workloads. We resorted to populating the database with the sample data provided by the application.

After populating the database, we issued HTTP requests to the application APIs and recorded the time taken to complete each request. For e-commerce applications, we replayed the user events in the order of the time each event occurred. For other applications, due to the absence of production traces, we randomly issued requests to the APIs with valid parameters from the database. Each experiment lasted for 15 minutes, and we regarded the first 5 minutes as the warm-up and only recorded the results in the last 10 minutes. The threshold of hot path identification was 20 in all experiments.

**Testbed.** We deployed a client server, a web server, and a database server separately for each experiment. Each machine has eight 3.4GHz vCPUs and 16GB RAM. The web server hosts applications via the OpenJDK 8 JVM, and we used MySQL 5.7.25 as the database. Unless otherwise specified, the network round trip time between machines is approximately 2.1 ms. The offline compiler operates on a different machine from the ones mentioned above.

## 9.2 Application Performance

This section quantifies the request latency reduction and throughput of the six applications in Table 1 with WeBridge. For latency experiments, we report the applications' end-to-end request processing time (the 50th/90th/99th percentile latency) with and without WeBridge. For the throughput experiments, we increase the number of clients until the total throughput reaches the peak.

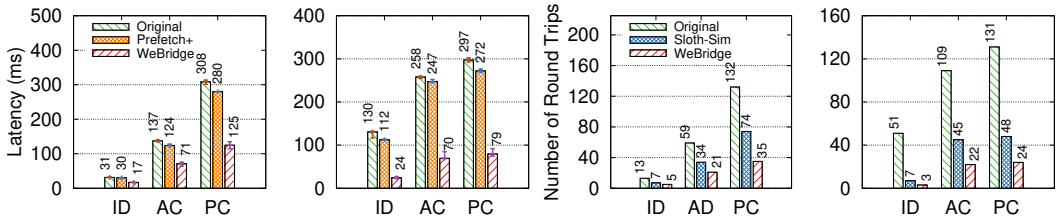


Fig. 12. Comparison with Prefetch+ and Sloth.

**Summary.** As shown in Figure 11, compared to the original application, WeBridge achieves up to 79.8% median latency reduction (with a geometric mean of 58.1%) and up to 2× peak throughput (with a geometric mean of 1.34×). The latency reduction mainly comes from the decreased number of round trips. Meanwhile, the throughput improvement is attributed to both the avoidance of repeatedly parsing and optimizing interactive SQL statements and the reduced duration of transactions operating contented data items. Table 2 shows the statistics of the generated stored procedures. On average, WeBridge generates 8.7 stored procedures for each API, and 1.1 of them are caused by inconvertible computations.

**E-Commerce Applications.** We evaluated three APIs that correspond to the three events found in the trace (§9.1): view product item details (ID), add products to shopping cart (AD), and purchase products (PC). The majority of the events are view products, causing the workload to be read-intensive.

Figure 11.a and Figure 11.b show the request processing time for each API. The result indicates that WeBridge significantly reduces end-to-end request processing latency compared with the original application, achieving 51%–79.8% median latency reduction. Figure 12.a and b show that WeBridge surpasses "Prefetch+" in all experiments, with 47.8%–78.9% decreased median latency. "Prefetch+" falls behind WeBridge because (i) the computations in both applications take far shorter time than the database communicating time, and (ii) dependent queries cannot be fetched in a single round trip, resulting in little overlapping opportunities.

The latency reduction achieved by WeBridge is largely attributed to the decreased number of round trips. Figure 12.c and d compare the average number of round trips for each API in the original application, Sloth, and WeBridge. The result shows that WeBridge efficiently saves round trips, with up to 94.1% reduction compared with the original application. In Shopizer, it reduces round trips by 61.5%–73.5%, while in Broadleaf, it saves between 79.8% to 94.1% round trips. Furthermore, WeBridge outperforms Sloth, with up to 57.1% more round trip reduction. This is because Sloth cannot batch queries with dependencies, while both Shopizer and Broadleaf have a lot of dependent queries. For example, 31% (4 out of 13) queries in Shopizer's ItemDetails API depend on other queries. As WeBridge if free of this limitation, it surpasses Sloth in all our experiments.

We next compare the throughput of the original applications with WeBridge-optimized ones. Figure 11.c and f show that WeBridge achieves similar peak throughput with the original applications with prepared statements enabled. However, WeBridge achieves peak throughput with fewer clients. This is primarily because the CPU and memory resources in both the web server and the database server are not the bottleneck when the number of clients is low. In these scenarios, network latency is the main bottleneck for throughput. As a result, the lower latency in WeBridge leads to a higher throughput compared with the original applications. As the number of clients increases, the database server's CPU becomes saturated, and the total throughput remains relatively stable regardless of the increasing number of clients.

**Blogging Applications.** For Sagan, we evaluated the APIs for viewing the details of the blog posts (BD) and creating/modifying blog posts (NP and EP). For My-Blog, we evaluated the APIs for

browsing the index page (GI), viewing the details of the blog posts (BD), and creating/modifying blog posts (EB for My-Blog). Mixing these APIs forms a write-intensive workload.

Figure 11.c and d report the request latencies of Sagan and My-Blog. The result shows that WeBridge can reduce the median latency of the original applications by 33.5%-67.5%. Figure 11.i and j report the throughput. WeBridge achieved 2× and 1.7× peak throughput of Sagan and My-Blog, respectively. The increased throughput of WeBridge is gained by reducing the time transactions hold locks. Take My-Blog as an example; the main bottleneck comes from the EB API, where the application updates the content and category of the blog. In this API, all the SQL statements run inside a single transaction, in which the database locks the record rows for the category being updated. When the number of clients is far more than the number of categories (at peak throughput), contention on rows in the category table greatly impacts the throughput. Since a transaction only releases its locks at commit time, transactions updating the same category have to be blocked and wait for its completion. The higher latency of the original application indicates a longer duration for which a transaction holds locks and blocks other conflicting transactions. WeBridge can reduce such blocking time by reducing the latency of the request, leading to a higher throughput.

**Forum Application.** For the Pybbs forum application, we evaluated the APIs for browsing the index page (GI), getting the hottest 100 forum topics (GT), and viewing its own user information (UI). The GI API contains SQL statements with join, order by, and aggregation, which are compute-intensive. Figure 11.c reports the request latencies of Pybbs. The result shows that Pybbs achieved a 46.9%-70.8% reduction in median latencies, due to the reduction of network round trips. Figure 11.g reports the throughput for Pybbs. At lower numbers of clients, the throughput linearly scales as the client number increases. The network latency becomes the bottleneck and WeBridge surpasses the original application. As the number of clients increases (to more than 10), the throughput of both the original and WeBridge-optimized applications increases to nearly the same. This is because the MySQL server's CPU has already been fully saturated, due to the execution of the compute-intensive SQL statements. Thus, the throughput cannot increase further as the number of clients grows.

**Configuration Management Application.** For the Apollo configuration management application, we evaluated the APIs for browsing the configuration management dashboard (GI), getting the details of managed applications (GA), and viewing the administrator information (UD). Figure 11.f shows the request latencies of Apollo. The result indicates that WeBridge reduces end-to-end API latencies by 52%-62.7%. Figure 11.i reports the throughput of Apollo. For both the original application and WeBridge-optimized application, the throughput scales linearly to the number of clients, and the main bottleneck is the network latency. When the number of clients increases to 64, the benchmark was not finished because the web server ran out of memory and was killed by the OS. This results in the final collapse of the throughput.

### 9.3 Factor Analysis

**Effectiveness of Speculative Execution.** We also evaluated Shopizer's APIs with speculative execution disabled, denoted as `Basic Alg`. The results, as shown in Figure 13, indicate that speculative execution reduces the end-to-end latency by 10.6%-32.2%. This reduction stems from eliminating the overhead of path condition evaluation in the database. Moreover, since path conditions could contain array operations, pruning these conditions helps WeBridge avoid splitting the dependency graph, thus generating fewer stored procedures. The ratio of cold paths is less than 1% and in these cases, speculative execution might fail. However, in such situations, WeBridge still reduces the request latency compared to the original application. This is because usually a front part of the SQL statements executed by the procedure is still correct, and WeBridge only falls back to normal execution for the rest of the SQL statements.

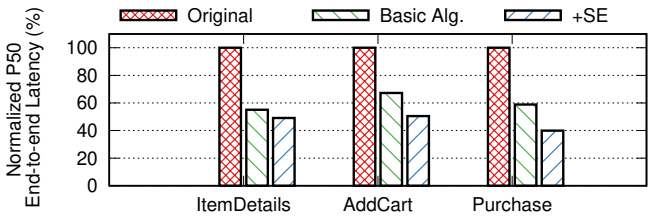


Fig. 13. The effect of WeBridge's speculative execution in Shopizer. X axis is the API name, Y axis is the Normalized End-to-end latency for the original application (Original), WeBridge without speculative execution (Basic Alg.) and WeBridge with speculative execution (+SE).

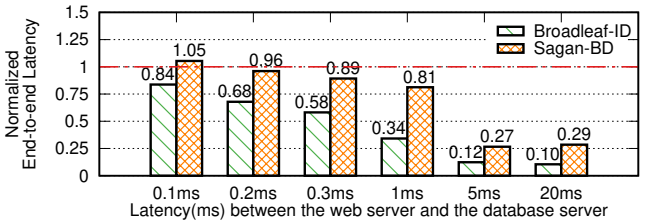


Fig. 14. API latency using WeBridge under different network delays, normalized to original application latency.

**Impact of Different Network Delays.** The improvement brought by WeBridge varies under different network settings. We selected two APIs, Broadleaf's ID and Sagan's BD, which respectively have the most and least latency reduction from the previous evaluation, and re-evaluated them under different network latencies. The results, shown in Figure 14, reveal that for Broadleaf's ID API, there is more than a 30% latency reduction even with a 0.2 ms network delay. For Sagan's BD API, there is a 19% latency reduction with a 1 ms network delay. When we increase the network delay to 20 ms, the latency reduction becomes more significant, reaching 71%. Note that, at a 0.1 ms network delay, Sagan's BD API is slower than the original application, due to WeBridge's runtime overhead.

**Runtime Overhead.** Figure 15 shows the the time consumed by the database server, network round trips, and the web server for three APIs in Shopizer. The majority of time is spent in the network communication, which is reduced significantly by WeBridge as expected. Also, the time spent in the database server is slightly reduced, from 12.5 ms to 11 ms, due to the absence of SQL parsing and planning. However, we observed that with WeBridge enabled, the application server consumes more time. On average, it increases from 1.8 ms (2% of total time) to 4.4 ms (9% of total time). This is because WeBridge needs to record request inputs as well as to validate the speculatively executed stored procedures.

## 10 Related Work

**Static Analysis on Web Applications.** Static analysis has been adopted to web applications for various purposes [28, 37, 38, 41, 50, 66, 76, 92, 105]. One way is using static program analysis techniques to obtain intermediate representations of the application and perform further analysis. Previous works such as Pysix and StatusQuo [37, 38] have employed static dependency analysis to construct an intermediate representation called the *partition graph*, which captures the dependencies between program statements. With *partition graphs*, these approaches partition the application into two parts: one executed on the web server and the other executed on the database server. This partitioning helps reduce the latency caused by round trips to the database. Meanwhile, DBridge [92] extracts the SQL statements in the application and prefetch their results to reduce

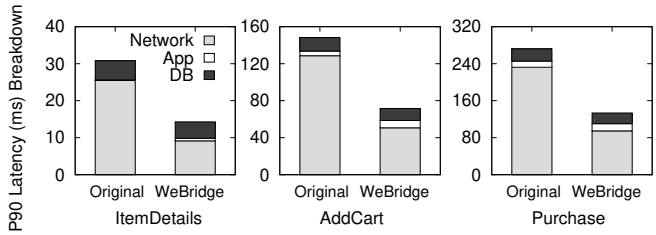


Fig. 15. Latency breakdown of three Shopizer APIs.

application latency. It used static analysis to construct a control-flow graph (CFG) of the application, then perform *query anticipability* analysis on the CFG to determine the earliest point each SQL statement could be issued.

**Concolic Execution.** Concolic execution has been extensively studied [33, 55, 65, 77, 96, 98, 110]. It is a program analysis technique that explores specific program paths. It associates each symbolic variable with a concrete value to guide program execution. Most previous works used it for bug hunting. After exploring an initial path, the concolic execution engine acquires the path conditions. It then negates a constraint in the path condition and employs the SMT solver [27, 44, 45] to find a satisfiable solution leading to another path. Unlike prior works, WeBridge employs concolic execution to extract program control and data dependencies.

**Web Application Performance Optimizations.** Several studies have focused on reducing network round trips between the web application server and the database by employing prefetching and batching [36, 39, 40, 78, 90, 92]. For instance, Sloth [39, 40] batches queries' execution through lazy evaluation. However, it cannot batch queries with dependencies, whereas WeBridge supports batching dependent queries into stored procedures. Some other works aim at reducing the database processing times in web applications. For example, QBS [42] utilizes query synthesis technique to transform the application code into more efficient query codes. Further, [107, 108] provide insights into designing web applications that leverage ORM frameworks to issue more efficient queries, and [34] proposes a bi-metric resource management approach to scale web applications more efficiently.

**Stored Procedure Optimizations.** Several methods have been proposed to improve the performance of procedural codes in DBMSs. Some utilize program analysis techniques to transform procedural code into more efficient declarative SQL [49, 56, 58, 85, 91]. For example, Aggify [56] rewrites cursor loops into aggregate queries, opening up new optimization opportunities like pipelining and computation push-down. Another line of research focuses on compiling interpreted executed procedure code into natively executed code [53, 68, 71–73, 99]. The compiled code is much more efficient than interpreted executed code and enables the application of existing code optimization techniques to further boost the execution efficiency. WeBridge's speculative execution (§7) is orthogonal to above optimizations, allowing the stored procedures generated by WeBridge to benefit from these existing optimizations.

## 11 Conclusion

We propose WeBridge, the first system that synthesizes stored procedures for large-scale real-world web applications by compiling only hot paths via concolic execution. Our evaluation result shows that the synthesized stored procedures can achieve up to 79.8% median latency reduction and up to 2× peak throughput.

## ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their valuable comments. This work is supported by the National Natural Science Foundation of China (No. 62272304 and 62132014).

## References


[1] 2020. Byte Buddy - runtime code generation for the Java virtual machine. https://bytebuddy.net/#/
[2] 2020. Java JDBC API. https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/
[3] 2020. MySQL 8.0 Reference Manual – SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT Statements. https://dev.mysql.com/doc/refman/8.0/en/savepoint.html
[4] 2020. PostgreSQL: Document: 13: SAVEPOINT. https://www.postgresql.org/docs/13/sql-savepoint.html
[5] 2020. The Rise of REST API. https://blog.restcase.com/the-rise-of-rest-api/
[6] 2020. Stored procedure - Wikipedia. https://en.wikipedia.org/wiki/Stored_procedure
[7] 2022. BroadleafCommerce. https://github.com/BroadleafCommerce/BroadleafCommerce
[8] 2022. Dart2js: Dart-to-javascript compiler. https://dart.dev/tools/dart2js
[9] 2022. J2OBJC; google developers. https://developers.google.com/j2objc?hl=en
[10] 2022. MyBatis. https://mybatis.org/mybatis-3/
[11] 2022. shopizer. https://github.com/shopizer-ecommerce/shopizer
[12] 2023. Better use of Java development community (forum) - pybbs. https://github.com/tomoya92/pybbs
[13] 2023. A blogging system implemented with spring-boot & thymeleaf & mybatis. https://github.com/ZHENFENG13/My-Blog
[14] 2023. Companies using Broadleaf Commerce. https://enlyft.com/tech/products/broadleaf-commerce
[15] 2023. Connector/J Reference. https://dev.mysql.com/doc/connector-j/8.1/en/connector-j-reference.html
[16] 2023. Extra Materials for WeBridge. https://anonymous.4open.science/r/webridge-materials-2D8A
[17] 2023. Known Users of Apollo. https://github.com/apolloconfig/apollo#known-users
[18] 2023. A reliable configuration management system - Apollo. https://github.com/apolloconfig/apollo
[19] 2023. Spring | Home. https://spring.io/
[20] 2023. The spring.io site and reference application - sagan. https://github.com/spring-io/sagan
[21] ActiveRecord. 2022. ActiveRecordApis. https://api.rubyonrails.org/classes/ActiveRecord/Base.html
[22] Frances E. Allen and John Cocke. 1976. A program data flow analysis procedure. *Commun. ACM* 19, 3 (1976), 137.
[23] Saswat Anand and Mary Jean Harrold. 2011. Heap cloning: Enabling dynamic symbolic execution of java programs. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 33–42.
[24] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static Analysis of Java Enterprise Applications: Frameworks and Caches, the Elephants in the Room. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 794âĂŞ807. https://doi.org/10.1145/3385412.3386026
[25] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2015. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1327âĂŞ1342. https://doi.org/10.1145/2723372.2737784
[26] Roberto Baldoni, Emilio Coppa, Daniele Cono DâĂŹElia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.
[27] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11) (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf Snowbird, Utah.
[28] Ivan T. Bowman and Kenneth Salem. 2007. Semantic Prefetching of Correlated Query Sequences. In *2007 IEEE 23rd International Conference on Data Engineering*. 1284–1288. https://doi.org/10.1109/ICDE.2007.368994
[29] Broadleaf BroadleafCommerce. 2022. BroadleafAddCart. https://github.com/BroadleafCommerce/ReactStarter/blob/3ea3b209064654f4a6bcd4882b994535d739832d/api/src/main/java/com/mycompany/api/endpoint/cart/ReactCartEndpoint.java#L157
[30] Broadleaf BroadleafCommerce. 2022. BroadleafCheckout. https://github.com/BroadleafCommerce/ReactStarter/blob/3ea3b209064654f4a6bcd4882b994535d739832d/api/src/main/java/com/mycompany/api/endpoint/checkout/ReactCheckoutEndpoint.java#L314
[31] Stefan Bucur, Johannes Kinder, and George Candea. 2014. Prototyping Symbolic Execution Engines for Interpreted Languages. *SIGPLAN Not.* 49, 4 (feb 2014), 239âĂŞ254. https://doi.org/10.1145/2644865.2541977
[32] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209âĂŞ224.
[33] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. 2020. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In *Annual Computer Security Applications Conference*. 746–759.

[34] Yinliang ZHAO Changpeng ZHU, Bo HAN. 2022. A bi-metric autoscaling approach for n-tier web applications on kubernetes. *Frontiers of Computer Science* 16, 3, Article 163101 (2022), 0 pages. https://doi.org/10.1007/s11704-021-0118-1

[35] Sarah Chasins, Alvin Cheung, Natacha Crooks, Ali Ghodsi, Ken Goldberg, Joseph E. Gonzalez, Joseph M. Hellerstein, Michael I. Jordan, Anthony D. Joseph, Michael W. Mahoney, Aditya G. Parameswaran, David A. Patterson, Raluca Ada Popa, Koushik Sen, Scott Shenker, Dawn Song, and Ion Stoica. 2022. The Sky Above The Clouds. *CoRR* abs/2205.07147 (2022). https://doi.org/10.48550/arXiv.2205.07147 arXiv:2205.07147

[36] Mahendra Chavan, Ravindra Guravannavar, Karthik Ramachandra, and S Sudarshan. 2011. Program transformations for asynchronous query submission. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 375–386.

[37] Alvin Cheung, Owen Arden, Samuel Madden, Armando Solar-Lezama, and Andrew C Myers. 2013. StatusQuo: Making Familiar Abstractions Perform Using Program Analysis.. In *CIDR*. Citeseer.

[38] Alvin Cheung, Samuel Madden, Owen Arden, and Andrew C. Myers. 2012. Automatic Partitioning of Database Applications. *Proc. VLDB Endow.* 5, 11 (jul 2012), 1471âĂŞ1482. https://doi.org/10.14778/2350229.2350262

[39] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. 2014. Sloth: Being Lazy is a Virtue (When Issuing Database Queries). In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 931âĂŞ942. https://doi.org/10.1145/2588555.2593672

[40] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. 2016. Sloth: Being Lazy Is a Virtue (When Issuing Database Queries). *ACM Trans. Database Syst.* 41, 2, Article 8 (jun 2016), 42 pages. https://doi.org/10.1145/2894749

[41] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2012. Inferring SQL queries using program synthesis. *arXiv preprint arXiv:1208.2013* (2012).

[42] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. *ACM SIGPLAN Notices* 48, 6 (2013), 3–14.

[43] Jong-Deok Choi and Harini Srinivasan. 1998. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools* (Welches, Oregon, USA) *(SPDT '98)*. Association for Computing Machinery, New York, NY, USA, 48âĂŞ59. https://doi.org/10.1145/281035.281041

[44] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *Proceedings of TACAS (LNCS, Vol. 7795)*, Nir Piterman and Scott Smolka (Eds.). Springer.

[45] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[46] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic Shared Memory Multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 85âĂŞ96. https://doi.org/10.1145/1508244.1508255

[47] doctrine. 2022. doctrine2ormdocumentation. https://www.doctrine-project.org/projects/doctrine-orm/en/2.13/index.html

[48] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. 2008. Execution Replay of Multiprocessor Virtual Machines. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Seattle, WA, USA) *(VEE '08)*. Association for Computing Machinery, New York, NY, USA, 121âĂŞ130. https://doi.org/10.1145/1346256.1346273

[49] Christian Duta, Denis Hirn, and Torsten Grust. 2019. Compiling pl/SQL away. *arXiv preprint arXiv:1909.03291* (2019).

[50] K. Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S. Sudarshan. 2016. Extracting Equivalent SQL from Imperative Code in Database Applications. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1781âĂŞ1796. https://doi.org/10.1145/2882903.2882926

[51] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[52] Roy T Fielding and Richard N Taylor. 2000. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Irvine.

[53] Craig Freedman, Erik Ismert, Per-Åke Larson, et al. 2014. Compilation in the Microsoft SQL Server Hekaton Engine. *IEEE Data Eng. Bull.* 37, 1 (2014), 22–30.

[54] Andy Georges, Mark Christiaens, Michiel Ronsse, and Koenraad De Bosschere. 2004. JaRec: a portable record/replay environment for multi-threaded Java applications. *Software: practice and experience* 34, 6 (2004), 523–547.

[55] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated whitebox fuzz testing.. In *NDSS*, Vol. 8. 151–166.

[56] Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. 2020. Aggify: Lifting the curse of cursor loops using custom aggregates. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 559–573.

[57] Surabhi Gupta and Karthik Ramachandra. 2021. Procedural Extensions of SQL: Understanding Their Usage in the Wild. *Proc. VLDB Endow.* 14, 8 (oct 2021), 1378âĂŞ1391. https://doi.org/10.14778/3457390.3457402

[58] Ravindra Guravannavar and S Sudarshan. 2008. Rewriting procedures for batched bindings. *Proceedings of the VLDB Endowment* 1, 1 (2008), 1107–1123.

[59] Gunnar Morling Hardy Ferentschik. 2022. HibernateValidator. https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/

[60] Peter Bhat Harkins. Mar. 2018. Lobste.rs access pattern statistics for research purposes. https://lobste.rs/s/cqnzl5/lobste_rs_access_pattern_statistics_for#c_hj0r1b

[61] Hibernate. 2022. https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html

[62] Hibernate. 2022. HibernateJavaDoc. https://docs.jboss.org/hibernate/orm/5.4/javadocs/

[63] Yigong Hu, Gongqi Huang, and Peng Huang. 2020. Automated reasoning and detection of specious configuration in large systems with symbolic execution. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 719–734.

[64] JavaSE8. 2022. Package java lang instrument. https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html

[65] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. 2009. jFuzz: A concolic whitebox fuzzer for Java. (2009).

[66] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 6–pp.

[67] Michael Kechinov. 2019. ECommerce behavior data from Multi Category Store. https://www.kaggle.com/datasets/mkechinov/ecommerce-behavior-data-from-multi-category-store

[68] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2209–2222.

[69] Kickoke, Mednizar, Med Nizar, Lokesh Gupta, Syed, Admin, Robin, Harpreet Kaur, Tim, Mohammed Amine, and et al. 2021. Statelessness in rest apis. https://restfulapi.net/statelessness/

[70] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. 2012. Efficient patch-based auditing for web application vulnerabilities. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 193–206.

[71] AndrÃĺ Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 197–208. https://doi.org/10.1109/ICDE.2018.00027

[72] AndrÃĺ Kohn, Viktor Leis, and Thomas Neumann. 2021. Making Compiling Query Engines Practical. *IEEE Transactions on Knowledge and Data Engineering* 33, 2 (2021), 597–612. https://doi.org/10.1109/TKDE.2019.2905235

[73] Konstantinos Krikellas, Stratis D. Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 613–624. https://doi.org/10.1109/ICDE.2010.5447892

[74] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection - Literature Review and Empirical Study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 507–518. https://doi.org/10.1109/ICSE.2017.53

[75] GREG LINDEN. 2006. Marissa Mayer at Web 2.0. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html/

[76] V Benjamin Livshits and Monica S Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis.. In *USENIX security symposium*, Vol. 14. 18–18.

[77] Rupak Majumdar and Koushik Sen. 2007. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 416–426.

[78] Amit Manjhi, Charles Garrod, Bruce M Maggs, Todd C Mowry, and Anthony Tomasic. 2009. Holistic query transformations for dynamic web applications. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 1175–1178.

[79] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 252–269.

[80] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*. 97–108.

[81] OpenJDK. 2022. Zero-assembler project. https://openjdk.org/projects/zero/

[82] Oracle. 2022. JavaClassloader. https://docs.oracle.com/javase/8/docs/api/java/lang/ClassLoader.html

[83] Admin OW2. 2022. Rubbos. https://projects.ow2.org/view/rubbos/

[84] Admin OW2. 2022. Rubis. https://projects.ow2.org/view/rubis/
[85] Kisung Park, Hojin Seo, Mostofa Kamal Rasel, Young-Koo Lee, Chanho Jeong, Sung Yeol Lee, Chungmin Lee, and Dong-Hun Lee. 2019. Iterative query processing based on unified optimization techniques. In *Proceedings of the 2019 International Conference on Management of Data*. 54–68.
[86] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H Lee, and Shan Lu. 2009. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 177–192.
[87] Corina S Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 179–180.
[88] Andrew Pavlo. 2017. What Are We Doing With Our Lives? Nobody Cares About Our Concurrency Control Research. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 3. https://doi.org/10.1145/3035918.3056096
[89] Dan Quinlan and Chunhua Liao. 2011. The ROSE source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, Vol. 2011. Citeseer, 1.
[90] Karthik Ramachandra, Mahendra Chavan, Ravindra Guravannavar, and S Sudarshan. 2014. Program transformations for asynchronous and batched query submission. *IEEE Transactions on Knowledge and Data Engineering* 27, 2 (2014), 531–544.
[91] Karthik Ramachandra, Kwanghyun Park, K Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of imperative programs in a relational database. *Proceedings of the VLDB Endowment* 11, 4 (2017), 432–444.
[92] Karthik Ramachandra and S Sudarshan. 2012. Holistic optimization by prefetching query results. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 133–144.
[93] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing user-defined aggregations using symbolic execution. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 153–167.
[94] Michiel Ronsse and Koen De Bosschere. 1999. RecPlay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS)* 17, 2 (1999), 133–152.
[95] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE Symposium on Security and Privacy*. 317–331. https://doi.org/10.1109/SP.2010.26
[96] Koushik Sen. 2007. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 571–572.
[97] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 488–498.
[98] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 263–272.
[99] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1907âĂŞ1922. https://doi.org/10.1145/2882903.2915244
[100] Spring. 2022. aopspring. https://docs.spring.io/spring-framework/docs/5.0.0.M5/spring-framework-reference/html/aop.html
[101] Spring. 2022. SpringTxnMng. https://docs.spring.io/spring-framework/docs/current/reference/html/data-access.html#transaction
[102] SQLALCHEMY. 2022. SQLALCHEMY2documentation. https://docs.sqlalchemy.org/en/latest/
[103] Ion Stoica and Scott Shenker. 2021. From Cloud Computing to Sky Computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 26âĂŞ32. https://doi.org/10.1145/3458336.3465301
[104] Steve Ebersole Vlad Mihalcea. 2022. hibernate join lazy. https://docs.jboss.org/hibernate/orm/6.1/userguide/html_single/Hibernate_User_Guide.html#fetching-LazyCollection
[105] Ben Wiedermann and William R. Cook. 2007. Extracting Queries by Static Analysis of Transparent Persistence. *SIGPLAN Not.* 42, 1 (jan 2007), 199âĂŞ210. https://doi.org/10.1145/1190215.1190248
[106] Sean Work. 2011. How Loading Time Affects Your Bottom Line. https://blog.kissmetrics.com/loading-time/
[107] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2017. Understanding database performance inefficiencies in real-world web applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, 1299–1308.

[108] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. 2018. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 800–810.

[109] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. 2018. Powerstation: Automatically detecting and fixing inefficiencies of database-backed web applications in ide. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 884–887.

[110] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. 745–761.

Received July 2023; revised October 2023; accepted November 2023