SQL for Deep Dynamic Analysis?

Mohammad R. Azadmanesh
Università della Svizzera italiana, Lugano, Switzerland
Mohammad.Reza.Azadmanesh@usi.ch

Matthias Hauswirth
Università della Svizzera italiana, Lugano, Switzerland
Matthias.Hauswirth@usi.ch

Abstract
If we develop a new dynamic analysis tool, how should we expose its functionalities? Through an interactive user interface, a DSL, a specific API, or in some other way? In this paper, we discuss how to use an already existing language familiar to most software engineers, SQL, to perform deep dynamic analyses. The goal is to explore the trade-off between expressiveness and ease-of-use. We use BLAST as the dynamic analysis tool and map its trace information to a relational database. We find that, even though SQL is expressive enough for deep analysis of program executions and information flow, it is not quite straightforward to express some of the queries software engineers might be interested in. However, it removes the burden of learning a new language from scratch, which could make it worthwhile as an option in some cases.

Categories and Subject Descriptors D.2.5 [Software Engineering]: Testing and Debugging—Diagnostics, Debugging aids, Tracing, SQL, BLAST

Keywords Dynamic analysis, SQL, query-based analysis, program trace

1. Introduction
Software developers use dynamic analysis for different purposes such as debugging, memory analysis, information flow analysis, and exploring the program behavior. Dynamic analysis tools make a trade-off between expressiveness and ease-of-use in their interfaces. On one hand are easy-to-use tools like traditional debuggers, where one can easily set breakpoints in the source code or add a watchpoints on program variables to understand program behavior. However, it can be cumbersome or impossible to find a particular occurrence of a program point or a particular instance of a class, and to navigate causal links between uses and definitions. On the other hand there are tools which sacrifice ease-of-use in favor of expressiveness. Usually, such tools come together with an expressive language that a software developer can use for precisely specifying a desired analysis. Examples of such tools are PQL [8], PTQL [3], or even ASM [2]. Even though one can perform deep and precise analysis using these tools, they are not quite easy to use for common developers, and they often require learning a new language.

Given the spectrum of approaches, we want to shine the spotlight on an approach in the middle of the spectrum: an approach that benefits from the ease-of-use of an already existing and familiar language and at the same time is expressive enough to perform some deep analysis. In particular, we believe it might be worthwhile investigating SQL as a query language for dynamic analysis. We prototype our idea on top of a low-level dynamic analysis framework for Java programs called BLAST [1]. SQL is a specific language for defining and manipulating data in a relational database management system (RDBMS). The relations in an RDBMS prescribe a rather rigid form as opposed to the complex data structures normally used for keeping information flow traces about a program execution.

By building our prototype we can see the consequences of encoding full-fledged information flow traces in a relational model. Specifically, we are interested in whether it would be easy to formulate the kinds of queries necessary for basic debugging (breakpoints, watchpoints), call stack analysis, heap analysis, and program slicing.

2. Data Model
Figure 1 presents the data model we use for representing Java execution traces in an RDBMS. A database can store multiple program traces (table Traces). A Trace is composed of a set of Events. Each Event is generated upon executing a Java bytecode. The Events of a given Thread in a given Trace are ordered by a virtual timestamp stored in their index attribute. The event_id attribute uniquely identifies an Event across all Traces and Threads. Attribute class_code specifies the kind of bytecode (as given by BLAST) the Event executed.
Each bytecode in Java belongs to a method Activation (act_id) and is executed in the context of a Thread (thread_id). An Activation has local variables (Locals), operand stack slots (StackSlots), and a return value (ReturnValueLocs). Attribute provider in Activations specifies which (invocation) bytecode Event created this Activation.

BLAST uses a use-def model to represent the Values used/defined by each Event. Each Event can use/define zero or more Values which reside in some kind of memory location. Each Value can be used by zero or more Events, but it can be defined only by a single Event. As a result, we introduce a junction table called Uses to represent the many-to-many relationship between an Event and its used Values, whereas for the defined Values, we use the attribute provider in Values to specify the event which provided that value. Attribute location_id in Values specifies the memory location to which the Value is bound by referring to the attribute id of MemoryLocs.

The following tables represent the different kinds of memory location available in Java: ArrayElements, IFields (instance fields), CFields (static fields), StackSlots (operand stack slots), Locals, ReturnValueLocs, and PseudoLocs (pseudo-locations for control-dependencies). The attribute location_id in each of these tables can be used as a foreign key to join these tables with MemoryLocs.

3. Debugging

3.1 Breakpoint

Traditional debuggers allow a programmer to inspect the runtime state of a program step by step starting from a breakpoint. In this section, we show how to perform a typical debugging scenario using SQL on BLAST’s data model.

Setting a breakpoint boils down to specifying a line number in source code, but it doesn’t specify which occurrence. The SQL query in Listing 1 shows how to find all the occurrences of a line number in a specific source file.

An occurrence of a specific source line can end up with several bytecode events. If we want to have a deeper anal-
1 SELECT e.*
2 FROM Events e, Activations a
3 WHERE e. source_line_no = @LINE
4    and e. act_id = a. act_id
5    and a. source_name = @SOURCE

Listing 1. Finding all occurrences of a source line

1 SELECT *
2 FROM Values v, Locals l
3 WHERE l. name = @LOCAL
4    and l. act_id = @ACT_ID
5    and l. location_id = v. location_id

Listing 2. Finding all the value changes of a specific memory location

Analysis, say finding the Nth occurrence, then there is not any straight forward query on the given data model. It requires bypassing N-1 set of bytecode events where there is at least an occurrence of a bytecode event belonging to another source line number between each two consecutive occurrences. This clearly is not a straightforward query in our model.

3.2 Watchpoint

We can introduce a watchpoint by specifying the memory location on which we would like to set the watchpoint, i.e., the name of the program variable, the context in which it is active, and the desired occurrence number. The context can be an activation record for local variables and stack slots, and an object/class for fields. The SQL query in Listing 2 shows how to find all the value changes of a specific memory location, e.g., a local variable named LOCAL in a specific activation record ACT_ID.

This query returns all the values assigned to this memory location in its lifetime. To specify a particular row, there is not a standard solution among the implementations of SQL in different RDBMSs. For example, in Oracle, we could use rownum, whereas in Postgres, we could use limit, offset.

3.3 Slicing

Program slicing prunes a program such that those parts that could not have contributed to the failure are ruled out. The slicing process normally starts from a slicing criterion. A slicing criterion for a program specifies a window for observing its behavior. Program slicing is a useful technique in debugging. Similar to program slicing, trace slicing determines all points in the execution that influenced a given point in time and space, but in terms of instruction occurrences in the trace, rather than instructions in the program. Given that BLAST provides program traces, we focus on how to slice a trace using SQL.

Finding a backward slice requires tracking the use-def dependencies of events recursively, so it demands a recursive query. The query in Listing 3 shows how to find the backward slice with the slicing criterion set to SC_EVENT_ID. It starts with the slicing criterion itself (line 2) and recursively adds the events providing the use set of already visited events to the slice (lines 4-9). It continues this process up until it reaches to a point where there is nothing to follow back, i.e., the provider is null (line 9).

3.4 Case Study

We now focus on a somewhat more realistic debugging scenario. Consider the code snippet in Listing 4, a buggy implementation of matrix multiplication in Java. To debug this program, we need to formulate a hypothesis and verify it with respect to the corresponding state of the program at runtime. Assume we run this program to multiply a 2*3 matrix by a 3*4 one, and we do not get the expected result.

For debugging, we could then formulate a hypothesis; for example the following hypothesis about one of the elements of the resulting matrix C.

\[
HYPOTHESIS 1. c_{13} = a_{10}b_{03} + a_{11}b_{13} + a_{12}b_{23}
\]

One trivial aspect about this hypothesis is that there have to be 6 array elements involved in computing the value of \(c_{13}\). We can verify our hypothesis by formulating a query to determine which array elements were indeed used in computing the value of \(c_{13}\). We break down this query into two steps:

**Step 1:** We first need to specify which element we would like to know the history about, i.e., the slicing criterion. The following query shows how to do this:
4. Heap Analysis

In Java, all objects and array reside in the heap. One of the useful analyses for programmers is to traverse the object graph of the heap snapshot corresponding to a particular point of the execution. The data model shown in Figure 1 provides the required relationships to traverse that object graph.

The first requirement for a heap analysis is to provide a heap snapshot on which one can apply a query. BLAST traces each step of program execution. Given the large size of a program trace, BLAST does not store a full snapshot of the memory state for each step, but it just keeps the delta of state modification upon executing each event. That said, we need to extract the state of the heap from a particular point of execution. In other terms, we need to find all reachable objects and the relationship among them at a particular point of execution. One way to do this is to start from a root node in the object graph and recursively traverse the reference fields.

However, a program can assign different values to the instance fields of an object during the lifetime of the object. We need to specify which state of an instance field corresponds to the point of execution we are aiming for. We break down the problem of traversing a heap graph into two steps:

Step 1: Starting from a root object with the id @OID, we find the last values for each one of its instance fields provided that the values are assigned before the time we would like to have the snapshot, say @TIME. The query in Listing 5 shows how to get the results for this step. The query first finds all values whose provider event occurred after @TIME. Finally, it applies an aggregate function (in this case, MAX(value_id)) on each combination of an object and one of its instance fields to find its last value satisfying all the aforementioned conditions. For reusability purposes, we cast this query as a stored procedure called LAST_VALUES_BEFORE(@OID, @TRACE, @TIME).

Step 2: Step 1 provided the desired state for only a single level in the object graph traversal. We can perform the same logic recursively to traverse all reference fields the same way up until we visit all the objects reachable from the root in that particular snapshot. SQL-99 introduces the syntax for writing recursive queries. Listing 6 shows the corresponding query. It starts with the last values for each instance field of the root object (Lines 2-3) and in each step of recursion, it adds the objects pointed at via the last values of the reference fields of the object (Lines 4-7). The query in Listing 6 shows how to get the results for this step. The query first finds all values related to the root object which have two references to other objects (the Link objects representing the head and tail of the list). Each Link object has two references to other Link objects (the LinkedList object which itself is a doubly-linked list of Student objects). The list itself is represented by a LinkedList object which has two references to Link objects representing the head and tail of the list. Each Link object has two references to other Link objects (the
WITH RECURSIVE Traverse (n) AS (  
    SELECT object_id, instance_field_id,  
    last_value_id  
    FROM LAST_VALUES_BEFORE(OID, TRACE, TIME) 
    UNION  
    SELECT l.object_id, l.instance_field_id,  
    l.last_value_id  
    FROM Values v, Traverse t,
    LAST_VALUES_BEFORE(v.object_id, TRACE, TIME) l  
    WHERE v.value_id = t.last_value_id  
) 
SELECT * FROM Traverse;

Listing 6. Traversing an object graph from a root object

WITH RECURSIVE StackTrace AS (  
    SELECT a.*  
    FROM Activations a, Events e  
    WHERE e.act_id = a.act_id  
    and e.event_id = TIME  
    UNION  
    SELECT a.*  
    FROM Activations a, Events e,  
    StackTrace s  
    WHERE s.provider = e.event_id  
    and a.act_id = e.act_id  
) 
SELECT * FROM StackTrace;

Listing 8. Extracting stack trace for a particular point of execution during program execution. The data model presented in Figure 1 provides enough information to recover the state of the call stack at any point of interest. The query in Listing 8 shows how to retrieve the state of the stack at a particular point (TIME). It is again a recursive query, starting from the activation record which was active while our event of interest occurred (Lines 2-5). It finds the invocation event that created the activation record already visited and recursively finds its enclosing activation record and so on (Lines 7-11).

6. Lessons Learned

Using SQL enables programmers to formulate deep dynamic analysis queries without the need to learn a new language. However, this does not come without challenges:

1. Keeping data in a normalized database causes lots of joins, even for very simple analyses. This can be cumbersome.
2. Most of the analyses performed depend on the support for SQL-99 and in particular, recursive queries. This is not something that all RDBMSs support. Writing the queries without using recursive queries would be quite complicated and confusing.
3. Most of the built-in functions of SQL are not standard and different implementations in different RDBMSs have different syntax for that (see Section 3.2). This limits portability of queries between different RDBMSs.
4. Program traces are usually large. We have not evaluated performance yet, but we expect queries to be slow.
5. Even though SQL is a familiar language, it is not quite straightforward to write deep queries for dynamic analysis. It requires some time for the programmer to get familiar with the data model to issue deep queries correctly.
6. Parameterized stored procedures seem quite useful for dynamic analysis given the similarity of analyses one may need to perform during a session.
7. Related Work

There is a large body of work on dynamic analysis tools with a specific query language.

PTQL [3] is a query language that performs on program traces online. It instruments the programs at appropriate points to gather the information required for satisfying the query. It uses a relational query model on a schema composed of only two relations. Our approach is an offline approach on a database schema with more details.

PQL [8] is a query language that allows programmers to verify design rules of an application. The focus of PQL is to find abstractions in the form of stored procedures might help to provide answers to the questions raised by the programmer. It uses a relational query model on a schema composed of only two relations. Our approach is an offline approach on a database schema with more details.

BLAST [1] is a dynamic analysis tool for Java programs at the bytecode level. Its low-level instrumentation provides a thorough information flow analysis on program traces. It uses the dynamic slicing to track how objects flow through the program, how methods get called, and fields get accessed, whereas the data model we discussed can be used for deeper analysis.

Lencevicius et. al. [6] implemented a query-based debugger for exploring large object spaces to verify the relationships among objects. This technique is confined to a single state of the program whereas our approach targets the whole program execution.

FQL [9] is a query-based debugger performing on heap snapshots. It is confined to a single heap snapshot, whereas our approach is applied to a full history of the heap.

On the other hand, there are dynamic analysis tools requiring a general purpose language to implement the actual analysis.

DiSL [7] is a framework for bytecode instrumentation based on aspects. A programmer can perform an analysis by writing instrumentation code. The framework weaves the instrumentation code at appropriate locations in the bytecode array.

ASM [2] is a Java bytecode engineering library with two types of APIs, core and tree API.

BLAST [1] is a dynamic analysis tool for Java programs at the bytecode level. Its low-level instrumentation provides a thorough information flow analysis on program traces.

Finally, Whyline [4, 5] is an interrogative debugger that allows the programmer to ask why did and why didn’t questions about the program execution. It uses the dynamic slicing to provide answers to the questions raised by the programmer.

8. Conclusion

Based on our preliminary experience, we believe that using SQL could be an interesting compromise between expressiveness and ease-of-use for deep dynamic analysis. Our BLAST-based data model allows the formulation of deep analyses in a somewhat intuitive way, and many developers already are familiar at least with the simpler parts of SQL. However, the queries become quite verbose, especially due to the many joins required to navigate across all the involved tables. Moreover, most meaningful queries are recursive, a feature of SQL that developers might not be familiar with and that not all DBMSs support. Providing a clean set of abstractions in the form of stored procedures might help to get a user experience that is not too different from that of less expressive dynamic analysis DSLs.

Acknowledgments

The first author was funded by Swiss National Science Foundation grant 200021_135245.

References


