Database Application Developer Tools Using Static Analysis and Dynamic Profiling

Surajit Chaudhuri, Vivek Narasayya, Manoj Syamala

Microsoft Research
{surajitc,viveknar,manojsy}@microsoft.com

Abstract

Database application developers use data access APIs such as ODBC, JDBC and ADO.NET to execute SQL queries. Although modern program analysis and code profilers are extensively used during application development, there is a significant gap in these technologies for database applications because these tools have little or no understanding of data access APIs or the database system. In our project at Microsoft Research, we have developed tools that: (a) Enhance traditional static analysis of programs by leveraging understanding of database APIs to help developers identify security, correctness and performance problems early in the application development lifecycle. (b) Extend the existing DBMS and application profiling infrastructure to enable correlation of application events with DBMS events. This allows profiling across application, data access and DBMS layers thereby enabling a rich class of analysis, tuning and profiling tasks that are otherwise not easily possible.

1 Introduction

Relational database management systems (DBMSs) serve as the backend for many of today's applications. Such *database applications* are often written in popular programming languages such as C++, C# and Java. When the application needs to access data residing in a relational database server, developers typically use data access APIs such as ODBC, JDBC and ADO.NET for executing SQL statements and consuming the results. Application developers often rely on integrated development environments such as Microsoft Visual Studio or Eclipse which provide a variety of powerful tools to help develop, analyze and profile their applications. However, these development environments have historically had limited understanding of the interactions between the application and the DBMS. Thus a large number of security, correctness and performance issues can go undetected during the development phase of the application, potentially leading to high cost once the application goes into production.

In this paper, we summarize the key ideas, techniques and results of our project at Microsoft Research to develop tools for database application developers that leverage static program analysis [3] as well as dynamic profiling [2] of the application at runtime. We first discuss the framework for statically analyzing database application binaries to automatically identify security, correctness and performance problems in the database application. Our idea is to adapt data and control flow analysis techniques of traditional optimizing compilers

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

by exploiting semantics of data access APIs and the database domain to provide a set of analysis services on top of the existing compiler. These services include: (a) Extracting the set of SQL statements that can execute in the application. (b) Identifying properties of the SQL statements such as tables and columns referenced. (c) Extracting parameters used in the queries and their binding to program variables. (d) Extracting properties of how the SQL statement results are used in the application. (e) Analyzing user input and their propagation to SQL statements. Using the above services, we have built vertical tools for: detecting SQL injection vulnerability, extracting the SQL workload from application binary, identifying opportunities for SQL query performance optimizations, and identifying potential data integrity violations.

While such static analysis can greatly aid database application developers, many issues in the application can only be detected at runtime. For example, if there is a deadlock in the DBMS, detecting which tasks in the application are responsible for causing that deadlock today can be non-trivial despite the availablity of both application and database profiling tools. The key reason why such tasks are challenging is that the *context of an application* (threads, functions, loops, number of rows from a SQL query actually consumed by the application, etc.) and the *context of the database server* when executing a statement (duration of execution, deadlocks, duration for which the statement was blocked, number of rows returned etc.) cannot be easily correlated with each other. We discuss an infrastructure that can obtain and correlate the appropriate application context with the database context, thereby enabling a class of development, debugging and tuning tasks that are today difficult to achieve for application developers. We conclude with a discussion of some open issues and future work.

2 Motivating Scenarios

We discuss a set of motivating scenarios around security, correctness and performance of database applications that can potentially be detected via static analysis and/or dynamic profiling.

SQL injection vulnerability: A well known example of such a security problem is SQL injection vulnerability. Applications that execute SQL queries based on user input are at risk of being compromised by malicious users who can inject SQL code as part of the user input to gain access to information that they should not. Detecting SQL injection vulnerability at application development time can help developers correct the problem even before the application is deployed into production, thereby avoiding expensive or high profile attacks in production (e.g. [4]).

Database integrity constraint violations: In many real-world applications, certain database integrity constraints are enforced in the application layer and not the database layer. This is done for performance reasons or to avoid operational disruption to an application that has already been deployed. For example, in a hosted web service scenario the DBA might be reluctant to pay the cost of altering an existing table of a deployed application. In such scenarios, given a database constraint such as $[Products].[Price] > 0$ as input, it would be useful if we could automatically identify all places in the application code where the *Price* column can potentially be updated, and add an assertion in the application code to verify whether the constraint is honored.

Enforcing best practices: There can be correctness or performance problems due to the way the queries are constructed or used in the application. For example, there can be mismatch between the data type used in the application (e.g. *int*) and the data type of the column in the database (*smallint*). Such a mismatch is not detected by todays application development tools, which can lead to unexpected application behavior at runtime. Development teams need automatic support for enforcing a set of best practices in coding (e.g. - no SELECT * queries or always use the ExecuteScalar() ADO.NET API for queries that return scalar values). While code analysis tools like FxCop [5] aid to an extent for best practices, these tools have no database and data access specific domain knowledge to support the above kind of analysis.

Root-causing deadlocks in the DBMS: Certain database application issues can only be detected at runtime. Consider an application that executes two concurrent tasks each on behalf of a different user. Each task invokes certain functions that in turn issue SQL statements that read from and write to a particular table in the database.

Suppose a bug causes SQL statements issued by the application to deadlock with one another on the server. The database server will detect the deadlock and terminate one of the statements and unblock the other. This is manifested in the application as one task receiving an error from the server and the other task running to completion normally. Thus, while it is possible for the developer to know that there was a deadlock (by examining the DBMS profiler output or the server error message of the first task) it is difficult for the developer to know which function from the other task issued the corresponding statement that caused the server deadlock. In general, having the ability to identify the application code that is responsible for the problem in the database server can save considerable debugging effort for the developer.

Suggesting query hints: By observing usage of data access APIs of the application, it may be possible to improve performance via use of query hints. For example, it is common for applications to execute a query that returns many result rows but not actually consume all the results, e.g. because user actions drive how many results are viewed. In most database systems, by default the query optimizer generates a plan that is optimized for the case when all N rows in the result set are needed. If only the top k rows are required instead ($k < N$), the optimizer can often generate a much more efficient plan. Such a plan can be generated by providing a "FAST k" query hint. The important point to note is that the information about what value of k (number of rows consumed by the application) is appropriate is available only by profiling the application context.

3 Solution Overview

The tool can operate in two modes. In the static analysis mode (see Figure 1) the tool performs custom static analysis on the input binary as explained in [3]. The output is a set of potential security, performance and correctness problems in the application pertaining to use of the database. In the dynamic analysis mode (see Figure 2) the tool instruments the application binary and converts it into an event provider. Once instrumented, the developer launches the application after turning on tracing for all the three event providers: (1) Microsoft SQL Server tracing (2) ADO.NET tracing and (3) Instrumented events from the application. This allows events containing both application context and database context to be logged into the ETW [6] event log. The key post-processing step is done by our Log Analyzer that correlates application and server events using a set of matching techniques as explained in paper [2].

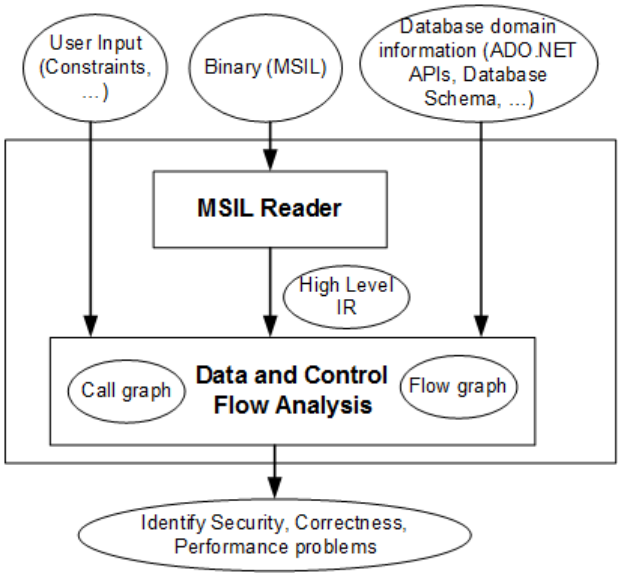

Figure 1: Overview of architecture of static analysis tool.

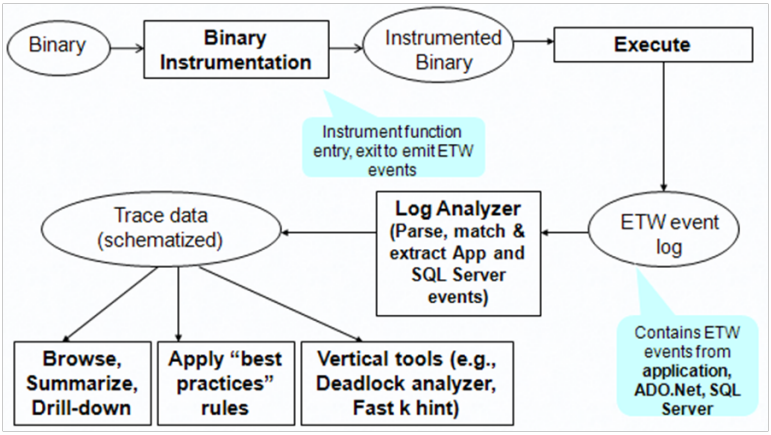

Figure 2: Overview of architecture of dynamic profiling tool.

Static Analysis: Our implementation of the static analysis tool relies on the Phoenix compiler framework [1]. We rely upon Phoenix to: (1) Convert the application binary in Microsoft Intermediate Language (MSIL) into an intermediate representation (IR) that our analysis operates upon. (2) Iterate over function unit(s) within the binary. (3) Provide the flow graph in order to iterate over basic blocks within a function unit. (4) Iterate over individual instructions in the IR within a basic block. (5) Provide extensions to dynamically extend the framework types like function units and basic blocks. (6) Provide a call graph that represents the control flow across function units and in case of dynamic analysis. (7) Instrument the binary so that it turns into a provider of ETW events. We extend this framework to develop the following primitives.

- *Extract SQL:* Given a function in the program binary, this primitive returns a set of SQL statement handles. A handle is a unique identifier that is a (*line number*, *ordinal*) pair in that function. It represents a SQL statement that can execute at that line number.

- *Identify SQL properties:* Given a *handle* to a SQL statement, this primitive returns properties of the SQL statement such as the SQL string, number and database types of columns in the result of the SQL statement, tables and columns referenced in the statement, and optimizer estimated cost.

- *Extract Parameters:*. Given a *handle* to a SQL statement this primitive returns the parameters of the statement along with the program variable/expression that is bound to that parameter, and its data type in the application.

- *Extract Result Usage:* Given a *handle* to a SQL statement, this primitive returns properties of how the result set is consumed in the application. In particular, it returns each column in the result set that is bound to a variable in the program, along with the type of the bound program variable.

- *Analyze User Input:* Given a *handle* to a SQL statement this primitive identifies all user inputs in the program such that the user input value v satisfies a contributes to relationship to the SQL string of the statement. A contributes to relationship is defined as either: (a) v is concatenated into the SQL string. (b) v is passed into a function whose results are concatenated into the SQL string.

The "vertical" functionality such as identifying SQL injection vulnerabilities, workload extraction and detecting potential data integrity violations are built using the above primitives. For additional technical details we refer the reader to [3].

Dynamic Profiling: Once the binary has been instrumented, the developer can click through a wizard (exposed as an Add-In to Microsoft Visual Studio), which launches the application after turning on tracing

for all the three event providers: (1) Microsoft SQL Server tracing (2) ADO.net tracing and (3) Instrumented events from the application. This allows events containing both application context and database context to be logged into the ETW event log. The key post-processing step is done by our Log Analyzer module that correlates application and server events using a set of matching techniques [2]. This matching is non-trivial since today there is no unique mechanism understood both by ADO.Net and Microsoft SQL Server to correlate an application event with a server event. The above collection and matching enables us to bridge the two contexts and provide significant added value to database application developers via "verticals" such as root causing deadlocks and fast k query hints.

4 Evaluation

We first provide a few examples using screenshots showing the functionality of the tools, and then summarize the results of applying the static analysis tool on a couple of real-world applications. A screenshot of the output of static analysis by the tool is shown in Figure 3. The left hand pane shows the functions in the binary. The SQL Information grid shows the SQL string, the SQL injection status (UNSAFE in this example). It also shows the actual line number in the code where the user input (leading to this vulnerability) originated, and the line number where the SQL statement is executed.

Figure 4 shows another screenshot of the static analysis tool. In this scenario, the user specifies via input that they expect the database constraint $[Products].[Price] > 0$ to hold, where Products is a table and Price is a column in that table. The right pane displays: (1) The fully formed SQL statement and the line number in the application where the SQL can execute when the application is run (in this case an INSERT statement). (2) Information about the parameters that are bound to the SQL statement. These include the parameter name, the data type and the application variable that is bound to the SQL parameter. (3) The application constraint corresponding to the input data integrity constraint specified by the user and the line number where it should be added. In this example the constraints analysis pane shows that expression ($price1 > 0$), where $price1$ is an application variable, will enforce the database constraint $[Products].[Price] > 0$ if it is placed at line number 279 in the application code.

In Figure 5 we see a screenshot showing the result of the dynamic profiling. The output of the tool is the summary/detail view as shown in the figure. Developers can get a summary and detail view involving various counters from the application, ADO.NET and Microsoft SQL Server, navigate the call graph hierarchy and invoke specific verticals. The Summary view gives the function name, aggregate time spend in a function, how many times the function was invoked and aggregate time spend executing the SQL statement (issued by the particular function) in the database server. Today the Function, Exclusive Time and Number of Invocations counters can be obtained from profiling the application using application side profiling tools such as Visual Studio Profiler; however the SQL Duration is an example of our value-add since it merges in database context into the application context. Consider the function *ReadStuff* which issues a SQL call. From the Summary view the developer can determine that the function was called twice and the aggregate time it spend inside this function (across all instances) was 5019 *msec*. Out of the total time spend in the function, most of the time was spend executing SQL (5006 *msec*). The Detail view gives more information at a function instance level. The tool allows drill down to display attributes of all the statements that were issued under the particular instance of the function or statements that were issued under the call tree of the particular instance of the function. The attributes of the SQL statement that are displayed include counters like duration, reads, writes, and also data access counters like reads issued by the application, and the data access API type, corresponding to the SQL that was issued.

A sample output from the deadlock analysis vertical is shown in Figure 6. The Microsoft SQL Server Profiler trace produces a Deadlock Event which contains the wait-for graph that describes a deadlock. The graph contains the statements being executed that resulted in the deadlock as well as timestamp, and client

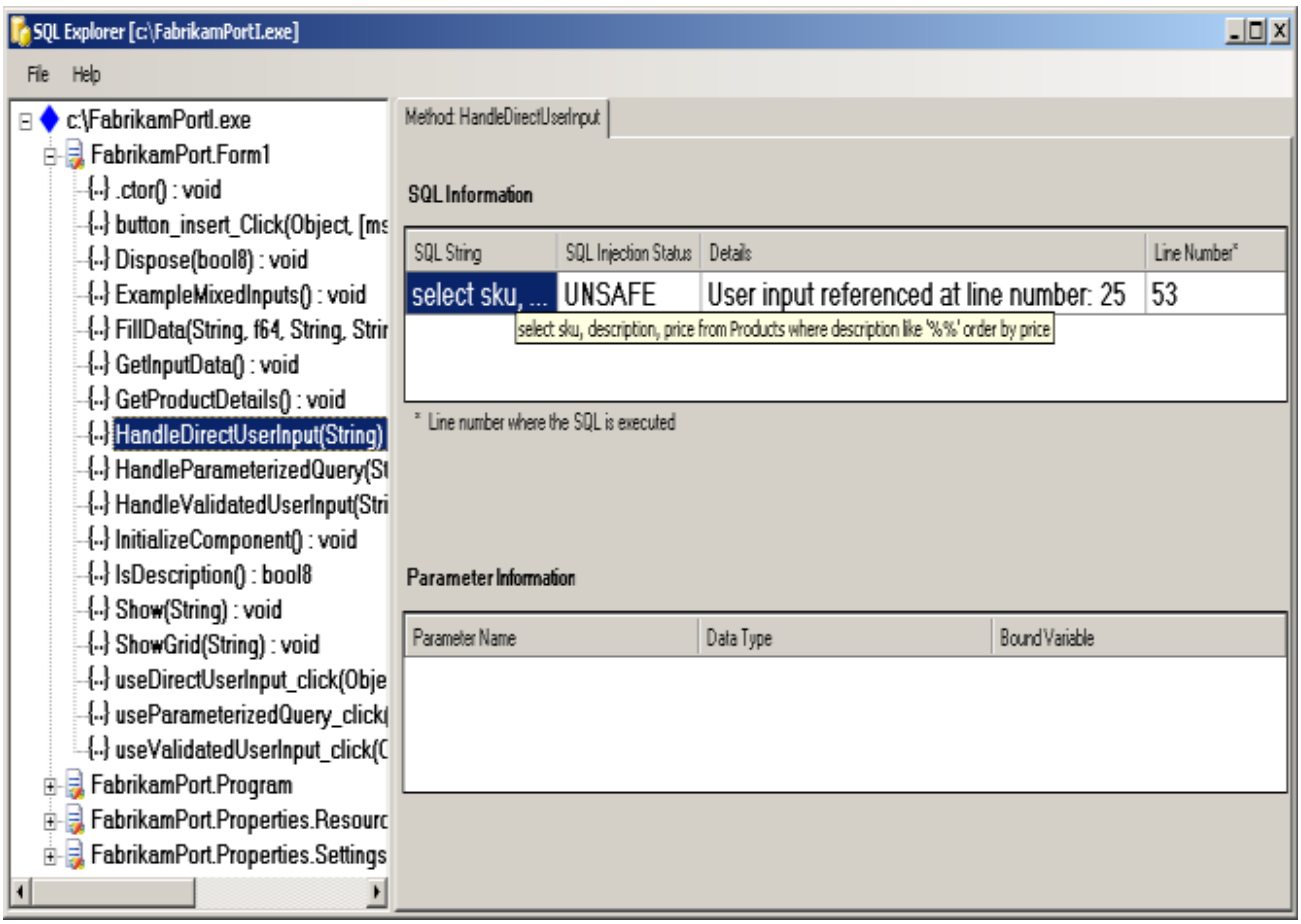

Figure 3: Output of the tool for SQL injection detection.

process id(s) information. The log analyzer extracts this information and stores it in the schematized application trace under the root node of the tree (as an event of type deadlock). For each such deadlock event, the deadlock analysis vertical finds the statements issued by the application that correspond to the statements in the deadlock event. Note that once we find the statement, we get all its associated application context such as function and thread. This can then be highlighted to the developer so they can see exactly which functions in their application issues the statements that lead to the deadlock.

We report briefly our experiences of running our static analysis tools on a few real world database applications: Microsofts Conference Management Toolkit (CMT): CMT [10] is a web application sponsored by Microsoft Research that handles workflow for an academic conference. SearchTogether [11]: An application that allow multiple users to collaborate on web search. For each application we report our evaluation of the Workload Extraction vertical. Our methodology is to compare the workload extracted by our tool with the workload obtained by manual inspection of the application code. The summary of results is shown in Table 5. The column *Total Num. of SQL statements* reports the number of that SQL statements that we were able to manually identify by examining the source code of the application. The column *Num. of SQL statements extracted* refers to the number of statements that were extracted by our static analysis tool. Along with the SQL statements we were able to extract parameter information as well. Thus, even though the actual parameter values are not known at compile time, we are able to extract syntactically valid queries. Thus it is possible, for example, to obtain a query execution plan for such queries. CMT and SearchTogether applications both mostly use parameterized stored procedures.

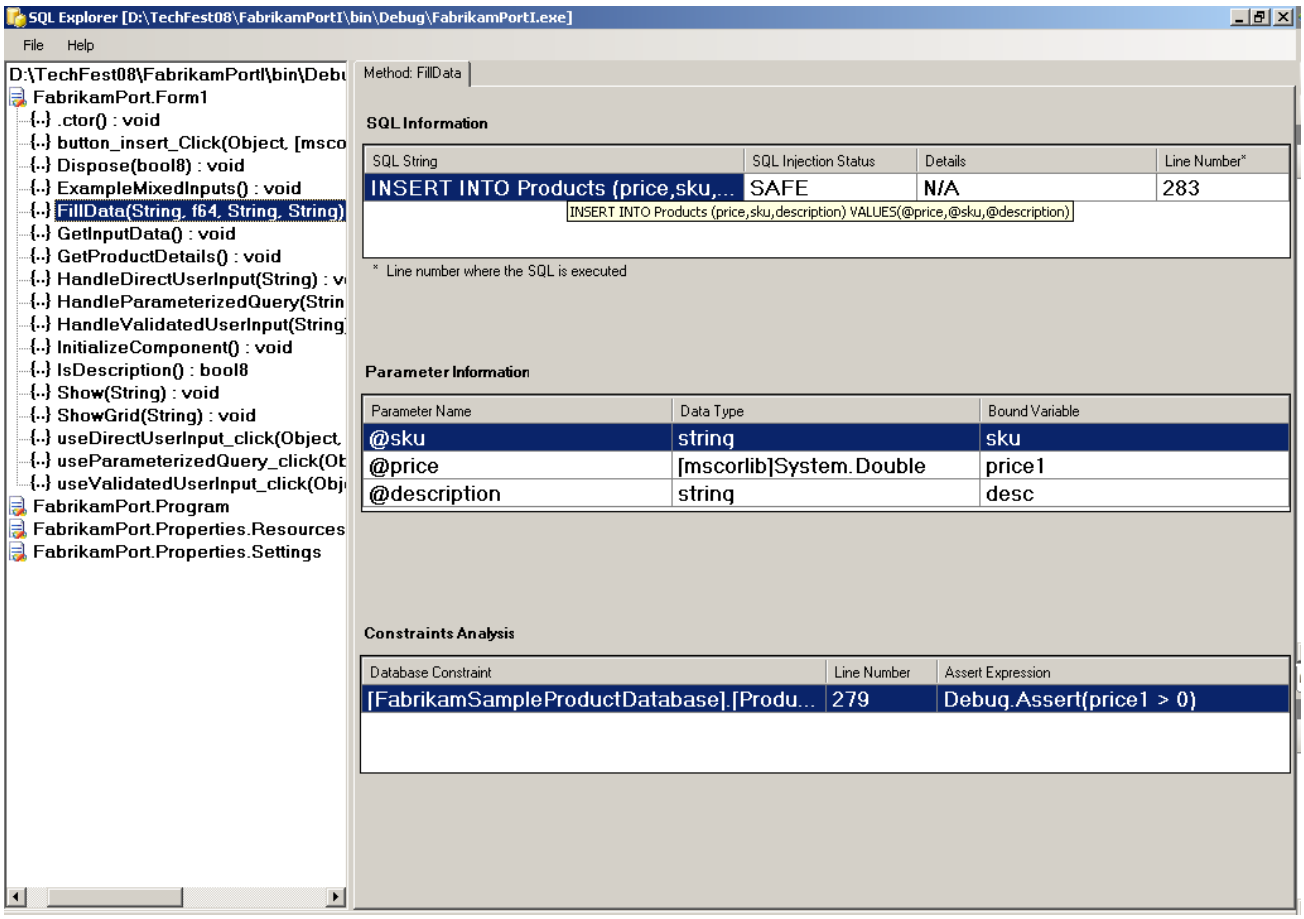

Figure 4: Detection of potential data integrity violation.

Application	Lines of code	Total Num. of SQL statements	Num. of SQL statements extracted
CMT	36,000+	621	350
SearchTogether	1,700	40	35

Table 5: Summary of results for workload extraction.

The cases where we were not able to extract SQL strings were due to the following reasons. First, there many ADO.NET APIs exposed by the providers that are used in these applications. Our current implementation does not cover the entire surface area of all the ADO.NET APIs. In some cases in SearchTogether, the *SQLCommand* object is a member variable of a class. The object is constructed in one method and referenced in another method. In this case, the global data flow analysis of our current implementation is not sufficient since the variable (the *SQLCommand* object in this case) is not passed across the two methods. Capturing this case requires tracking additional state of the *SQLCommand* object, which our current implementation does not. We also ran our SQL injection detection tool on all the three applications. We detected no SQL injection vulnerabilities in CMT and SearchTogether. In these applications user input is bound to parameters and executed as parameterized SQL.

5 Conclusion

In this paper we discussed how by exploiting our understanding of the semantics of data access APIs it is possible to detect a class of problems in the application through static program analysis. We also showed that the ability to

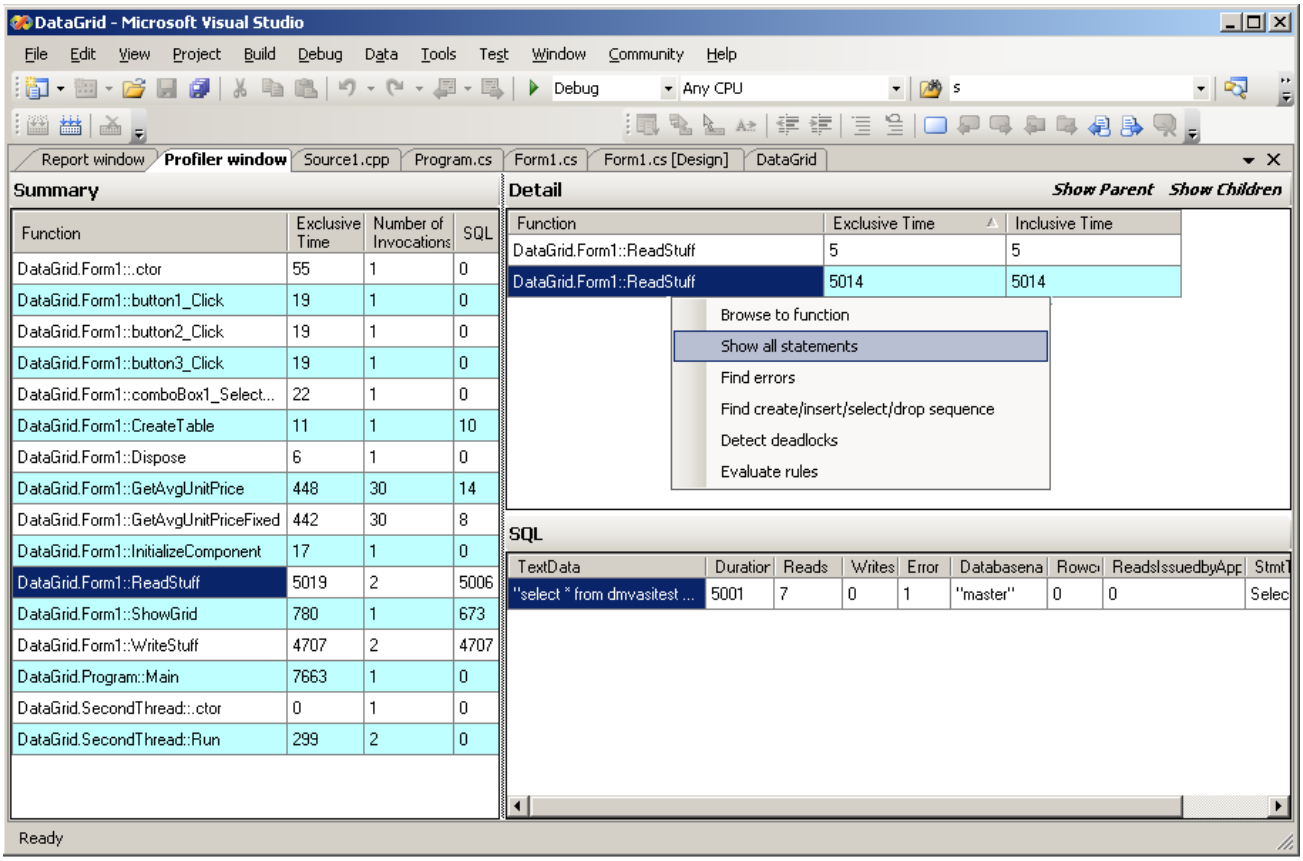

Figure 5: Summary and Detail views of Dynamic Profiling.

automatically profile and correlate application context and database context at runtime further enables expanding the class of issues that can be detected. We conclude with a discussion of a few open issues that are potential areas for future work.

- *Extending to applications in the cloud.* Applications running on cloud infrastructure often reference multiple services in addition to databases such as caching, queueing and storage. The applications use well defined APIs to interact with these services, similar to data access APIs for databases. Extending the techniques described here for such multi-tier applications by exploiting the semantics of these APIs could be valuable to application developers.

- *Analyzing SQL code.* Many database applications use stored procedures. Analyzing the SQL statements within these stored procedures using static analysis can help identify performance and security issues similar to those described here.

- *Profiling overhead.* Dynamic profilng can impose non-trivial overheads on performance both within the application and in the DBMS. While this may be acceptable during the application development phase, such overheads are unacceptable in a production setting. Thus optimizations that could limit the overheads of dynamic profiling could greatly broaden the scope of applicability of such tools to production use.

- *Correlating events in multi-tier applications.* Many database engines do not natively support propagation of a unique identifier that tracks a particular ”activity” (say for example a new order transaction in the application code) between the application and the database engine. Native support for such propagation

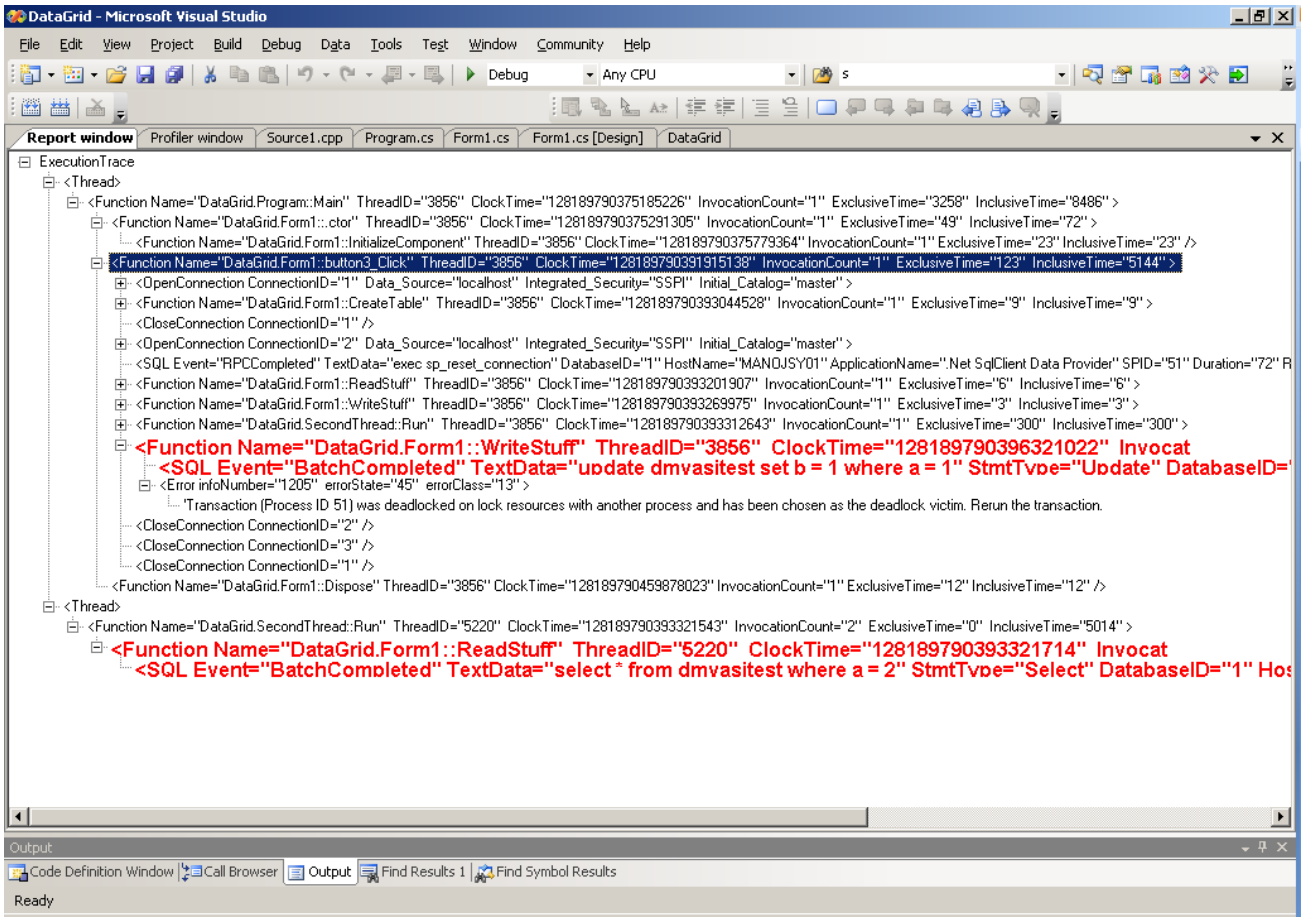

Figure 6: Output of deadlock analysis vertical.

of such an identifer would greatly ease the problem of correlating events logged at the application layer and events logged by the database server and eliminate the need to resort to approximate correlations such as time and textual similarity.

- *Automated performance optimizations*. Our focus has primarily been on *detecting* a range of security, performance and correctness problems in database applications so that developers can take appropriate actions to address the issues. An interesting related area that has received some attention more recently is automatically modifying application and/or database (SQL) code to improve program performance. Examples include automatically replacing imperative application code with equivalent declarative SQL code to improve efficiency [8] and automatically prefetching query results from the database based on understanding control flow in the application code [9].

References

[1] Phoenix compiler framework, http://en.wikipedia.org/wiki/Phoenix_compiler_framework

[2] Surajit Chaudhuri, Vivek R. Narasayya, Manoj Syamala: Bridging the Application and DBMS Profiling Divide for Database Application Developers. VLDB 2007: 1252-1262

[3] Arjun Dasgupta, Vivek R. Narasayya, Manoj Syamala: A Static Analysis Framework for Database Applications. ICDE 2009: 1403-1414

[4] United Nations vs. SQL Injections. http://hackademix.net/2007/08/12/united-nations-vs-sql-injections.

[5] FxCop: Application for analyzing managed code assemblies. http://msdn.microsoft.com.

[6] Event Tracing for Windows (ETW). http://msdn.microsoft.com.

[7] A. Aho, R. Sethi, and J. Ullman. Compilers. Principles, Techniques and Tools. Addison Wesley.

[8] Alvin Cheung, Owen Arden, Samuel Madden, Andrew C. Myers: Automatic Partitioning of Database Applications. PVLDB 5(11): 1471-1482 (2012).

[9] Karthik Ramachandra, S. Sudarshan: Holistic optimization by prefetching query results. SIGMOD Conference 2012: 133-144

[10] Microsoft Conference Management Service (CMT). http://msrcmt.research.microsoft.com/cmt/

[11] Microsoft SearchTogether application. http://research.microsoft.com/searchtogether/