Bridging the Application and DBMS Profiling Divide for Database Application Developers

Surajit Chaudhuri
Microsoft Research
One Microsoft Way
Redmond, WA 98052
+1 (425) 703-1938
surajitc@microsoft.com

Vivek Narasayya
Microsoft Research
One Microsoft Way
Redmond, WA 98052
+1 (425) 703-2616
viveknar@microsoft.com

Manoj Syamala
Microsoft Research
One Microsoft Way
Redmond, WA 98052
+1 (425) 703-5389
manojsy@microsoft.com

ABSTRACT
In today's world, tools for profiling and tuning application code remain disconnected from the profiling and tuning tools for relational DBMSs. This makes it challenging for developers of database applications to profile, tune and debug their applications, for example, identifying application code that causes deadlocks in the server. We have developed an infrastructure that simultaneously captures *both the application context as well as the database context*, thereby enabling a rich class of tuning, profiling and debugging tasks that is not possible today. We have built a tool using this infrastructure that enables developers to seamlessly profile, tune and debug ADO.NET applications over Microsoft SQL Server by taking advantage of information across the application and database contexts. We describe and evaluate several tasks that can be accomplished using this tool.

1. INTRODUCTION
Today, relational database management systems (RDBMSs) serve as the backend for many real-world, data intensive applications. These applications use programming interfaces such as ODBC [21], JDBC [16], ADO.NET [1], and OLE DB [24] to interact with the database server. When building these applications, application developers today use a variety of tools for understanding and fixing problems in their application. For example, development environments such as Microsoft Visual Studio [20] and Eclipse [9] provide profiling tools that allow developers to understand performance characteristics in their application code such as frequently invoked functions, time spent in a function etc. Of course, for database application developers such support is not sufficient since they use APIs to invoke DBMS functionality. Thus an important part of their application's execution happens inside the DBMS. DBMSs provide profiling and tuning tools of their own, e.g., Microsoft SQL Server Profiler [29], IBM DB2 Query Patroller [15] etc. These DBMS profiling tools give developers information about which SQL statements were executed against the server, the duration of each statement, read and writes performed by the statement, blocking activity etc. However, the information obtained from the development environment profiler and the DBMS profiler today remain as two islands of information that have little or no understanding of each other. This makes it difficult for database application developers to identify and fix problems with their applications as illustrated below:

Example 1. Detecting functions in the application code that caused a deadlock in the DBMS. Consider an application that has two threads, each executing a task on behalf of a different user. Each thread invokes certain functions that in turn invoke SQL statements that read from and write to a particular table T in the database. Consider a scenario where an intermittent bug in one of the threads 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 thread receiving an error from the server and the other thread running to completion as normal. 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 thread) it is difficult for the developer to know, for example, which function from the other thread issued the corresponding statement that caused the server deadlock. 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.

Example 2. Consider a table such as CUSTOMER (Name, Address, City, State, Zip, Comments). A common performance issue is that the application requests data (e.g. columns) from the server that are never used by the application. For example, the query "SELECT * FROM CUSTOMER" can be wasteful of server resources, particularly if the application only needs to use columns Name, Address from the CUSTOMER table. However, today neither the DBMS profiling tools (no awareness of which columns are actually used by the application) nor the profiling tools available in application development environments (no awareness of impact of the query on the resources consumed in the server) can identify this problem. Even doing an exact match search for the string ("SELECT *" in this example) in the application code might not help because the query string itself can be dynamically constructed by concatenating other strings. A tool that can identify such a problem would be useful to developers. Ideally, such a tool would even be able to suggest rewriting the query and quantify the associated performance improvement.

These examples highlight a significant gap that exists today in profiling technologies for database application developers. 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, duration for which the statement was blocked, number of rows returned etc.) remain uncorrelated with each other. We propose 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, as in the examples above. The prototype that we have developed extends the DBMS and application profiling infrastructure. It is built as an Add-In to Microsoft Visual Studio, and works for ADO.NET applications that execute against Microsoft SQL Server 2005. This integration makes it easy for a developer to invoke and interact with our tool from inside the Microsoft Visual Studio development environment. Our solution builds on the existing DBMS and application profiling infrastructure. There are three sources of information that we use which are generated when the application is executed:

- *Microsoft SQL Server tracing* [29]. This is the built-in profiling capability of the DBMS. There are several types of trace *events* that are exposed by the server for profiling. For example, *SQLStatementCompleted* is an event that is generated whenever a SQL statement completes. It contains attributes such as the SQL text, Duration of statement, Reads, Writes, Rows returned by the statement, etc. Another example is a *Deadlock* event, which is generated whenever the server identifies a deadlock and terminates a victim request. It contains attributes such as the text of the two deadlocking requests, which request was chosen as a victim etc.
- *ADO.NET tracing* [7]. This is the built-in profiling capability of the ADO.NET data access layer. Since the application uses the ADO.NET APIs to connect and interact with the DBMS, this tracing contains detailed information about how the application uses the APIs. For example, an event is generated each time the application opens a connection, executes a statement, consumes a row from the result set etc.
- *Application tracing*. We use binary instrumentation techniques [10] to inject code into the application. Since binary instrumentation is a post-compilation step, we don't need access to the application source code. When the application is run, the injected code emits certain events. For example, through appropriate code injection, we can make the application emit an event whenever a thread enters or leaves a function. The attributes of such an event include the identifier of the function, timestamp of entry, timestamp of exit etc. Another example of an event is whenever the application enters or leaves a loop.

The events obtained from each of these sources are written into an Event Tracing for Windows (ETW) [11] log file. ETW is an efficient and scalable logging infrastructure that allows different processes on a machine to generate and log traces in a uniform manner to the same trace session. An ETW event log is generated on each machine that involves either an application process or the DBMS server process. We then perform a key post-processing step over the ETW event log(s) to correlate the application and database contexts. The output of our post-processing is a single "view" where both the application and database profile of each statement issued by the application are exposed. For example, a row in this view contains information such as (ProcessId, ThreadId, Function Identifier, SQL statement, Rows returned, Rows consumed, Columns returned, Columns consumed, SQL duration, Blocked duration, Reads, Writes, Error code,…). This makes it possible to perform the application level debugging and tuning that SQL application developers need to do as in Examples 1 and 2.

Our current solution is targeted for use in development environments and is not engineered for use in production systems since the overheads introduced due to application instrumentation, ADO.NET tracing etc. can be significant. In Section 4 we briefly discuss how our solution can be adapted so it can potentially scale to production scenarios as well.

The rest of this paper is structured as follows. In Section 2 we provide more examples of tasks that are enabled by our infrastructure. In Section 3, we describe the functionality of the tool; and we present the technical details of our infrastructure in Section 4. Section 5 describes how we implement two interesting and non-trivial "vertical" tasks on top of our infrastructure. Section 6 describes our experiences using the tool with a few real applications. We also present an evaluation of the overheads incurred by application instrumentation. Section 7 discusses related work and we conclude in Section 8.

2. MOTIVATING SCENARIOS

In the introduction we presented two motivating examples for the infrastructure presented in this paper. In this section, we provide more scenarios. All of these scenarios share a common thread. These tasks cannot be achieved unless *both* application and database context information is available and can be correlated.

2.1 Suggesting "Fast k" query hint

Consider query Q10 from the TPC-H decision support benchmark [30], that returns for each customer the revenue lost due to items returned by the customer. The query is a join between four tables (customer, order, lineitem, and nation) and returns the customers in descending order of the lost revenue. The application code that consumes the results of the query may be written for example as:

```
conn = new SqlConnection(connString);
conn.Open();
SqlCommand cmd = new SqlCommand(cmdtext, conn);
cmdtext = @"SELECT .., SUM(l_extendedprice*(1-
        l_discount)) AS REVENUE...";
cmd.CommandText = cmdtext;
cmd.CommandType = CommandType.Text;
rdr = cmd.ExecuteReader();
while (rdr.Read()) {
        DisplayData(rdr);
        LostRevenue = rdr["REVENUE"];
        if(LostRevenue <= MINREVENUE)    break;
}
```

where the variable LostRevenue is bound to the column of the query which computes the revenue lost by the customer. In any given execution, the application may only consume a few rows from the entire result set of the query. In most DBMSs, when a query is executed, the query optimizer generates a plan that is optimized for the case when *all* rows in the result set are needed. If the above query returns many rows (say N) and the application consumes only *k* rows (*k* << N), then it may be beneficial to pass a query hint to the database server requesting that the plan be optimized for returning the top *k* rows quickly. For example, in Microsoft SQL Server this is achieved by using an OPTION (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 can only be obtained from the application context. Once this information is available, it is possible to perform interesting analysis (see Section 5.2) that can suggest to the developer if providing the hint will actually benefit the performance of the query.

2.2 Suggesting parameterization

A common performance problem on the server arises when applications do not parameterize their SQL. Consider a function in the application that when invoked with a parameter *p* (say having a value 23), executes a SQL statement with the value of that parameter: "Select … FROM R, S … WHERE … AccountId = 23". In another invocation, the parameter value could be different, and therefore in each invocation a different SQL text is submitted to the server. Thus, the server is required to treat each statement as requiring a potentially unique execution plan. Note that today's DBMSs have *auto parameterization* capabilities: however they typically apply only to very simple queries (such as single table selection queries). Thus, in the above example, the application can cause unnecessary compilations and inefficient usage of the DBMS procedure cache. Since the execution plans that are optimal for each of these instances may often be the same, it can be far more efficient for the application to parameterize its SQL: "Select … FROM R, S … WHERE … AccountId = @p", and pass in the parameter value via the data access APIs in the application. This tells the database server to treat different instances of that query as a single statement with a shared plan, which can dramatically reduce the compilation time as well as resources consumed in the procedure cache.

2.3 Identifying Opportunities for Bulk Operations

Consider a loop in the application code, shown below, inside which the application is inserting data into a table:

```
for (int i=0;i<MAXVALUE;i++) {
 // execute SQL statement   INSERT INTO T VALUES (…)
}
```

As written above, the code is inefficient since each time through the loop an INSERT statement is executed. A much more efficient way to achieve the same result is to use the bulk insert APIs of the data access layer, which takes advantage of batching. Note that the ingredients for identifying this problem is having the application context that a particular SQL statement is executed repeatedly inside a loop, and the database context to know that each instance is in fact an INSERT statement on a table T. It is then possible to put these pieces of information together to suggest a mapping to the bulk insert APIs.

2.4 Suggesting appropriate use of Data Access APIs

Many database application developers may be unaware of certain best practices for using data access APIs such as ADO.NET. For example, when executing a stored procedure, the best practice is to use the commandType.StoredProcedure option, passing the parameters using the AddParameters API [18]. This results in a Remote Procedure call (RPC) Event and is therefore efficient. However, a developer who is unaware of this may use the default commandType.Text option and pass in a string such as "exec my_sp 10". In this case, the database server gets a Language Event which needs to be parsed to find the stored procedure to be invoked, arguments to be passed etc. If the above code is executed many times, the performance improvement by using an RPC Event compared to Language Event can be significant. In this example, the key information from the application context is knowing that a Language Event was issued by the application *and* knowing from the database context that what was issued was in fact a stored procedure.

A second example of API usage arises when a query returns a scalar value (one row and one column) as the result. For such cases, ADO.NET provides an ExecuteScalar API [12] that is much more efficient that the more general ExecuteReader API. In this example, the application context is the fact that ExecuteReader API was used, and the database context is the fact that the query returns exactly one row and one column in its result set.

2.5 Identifying sequences for index tuning

Consider a *sequence* of SQL statements such as the one given below that is issued by an application:

```
CREATE TABLE R (…)
INSERT INTO R SELECT … FROM … WHERE …
SELECT … FROM R, T WHERE …
DROP TABLE R
```

The interesting characteristic of the above sequence is that table R is transient. It is possible that if the application were to issue a CREATE INDEX statement on table R after the INSERT statement but before the SELECT statement (there may be multiple such statements), the SELECT statement(s) could be speeded up enough to offset the cost of creating the index. The work in [3] describes a tool that given a sequence of SQL statements is able to recommend if indexes should be created or dropped in the sequence to reduce overall cost of the sequence.

The developer, who is the user of such a tool, still needs to be able to find such sequences in his/her application. This is necessary since applying the recommendations of the tool requires changes to the application code. This requires an understanding of the function(s) involved in producing the sequence of statements. Since a sequence can span function boundaries, extracting a sequence of SQL statements would require knowledge of the call graph structure of the application which is available only in the application side context.

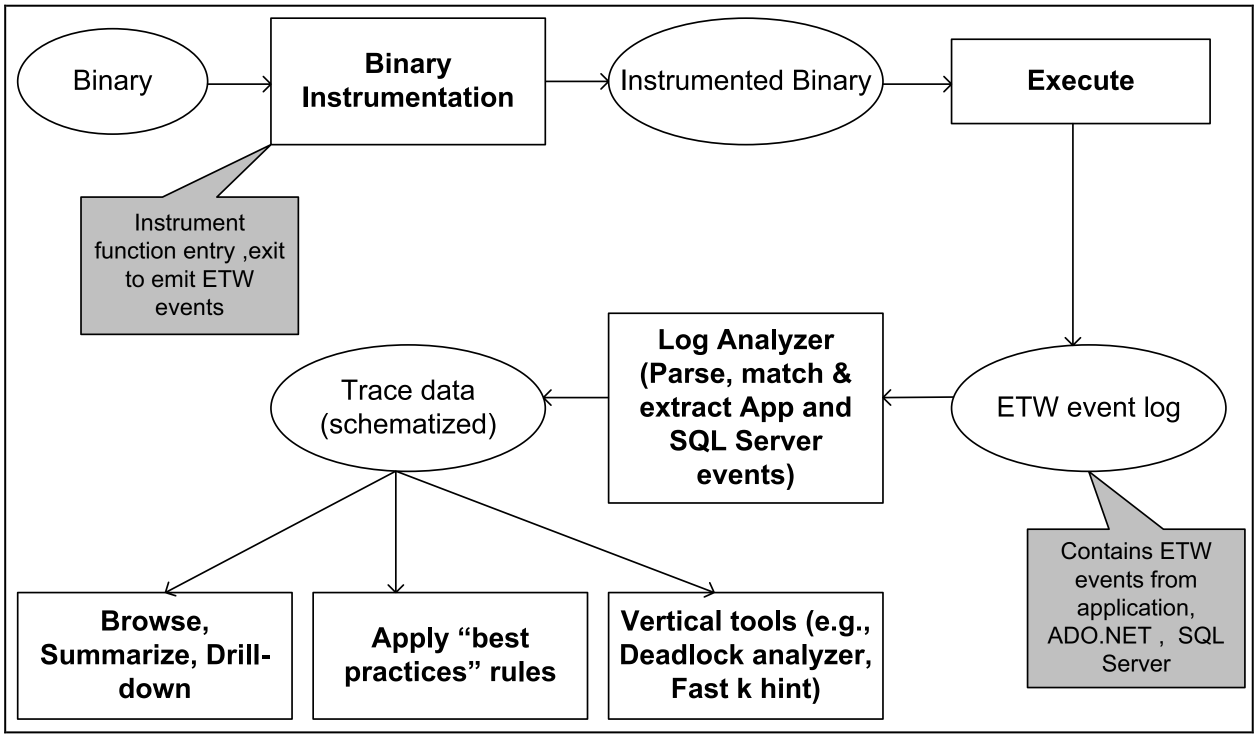

Binary
Binary Instrumentation
Instrumented Binary
Execute
Instrument function entry ,exit to emit ETW events
Log Analyzer (Parse, match & extract App and SQL Server events)
Trace data (schematized)
ETW event log
Contains ETW events from application, ADO.NET , SQL Server
Browse, Summarize, Drill-down
Apply "best practices" rules
Vertical tools (e.g., Deadlock analyzer, Fast k hint)

Figure 1. Architecture of our database application debugging and tuning tool.

3. ARCHITECTURE AND OVERVIEW OF THE TOOL

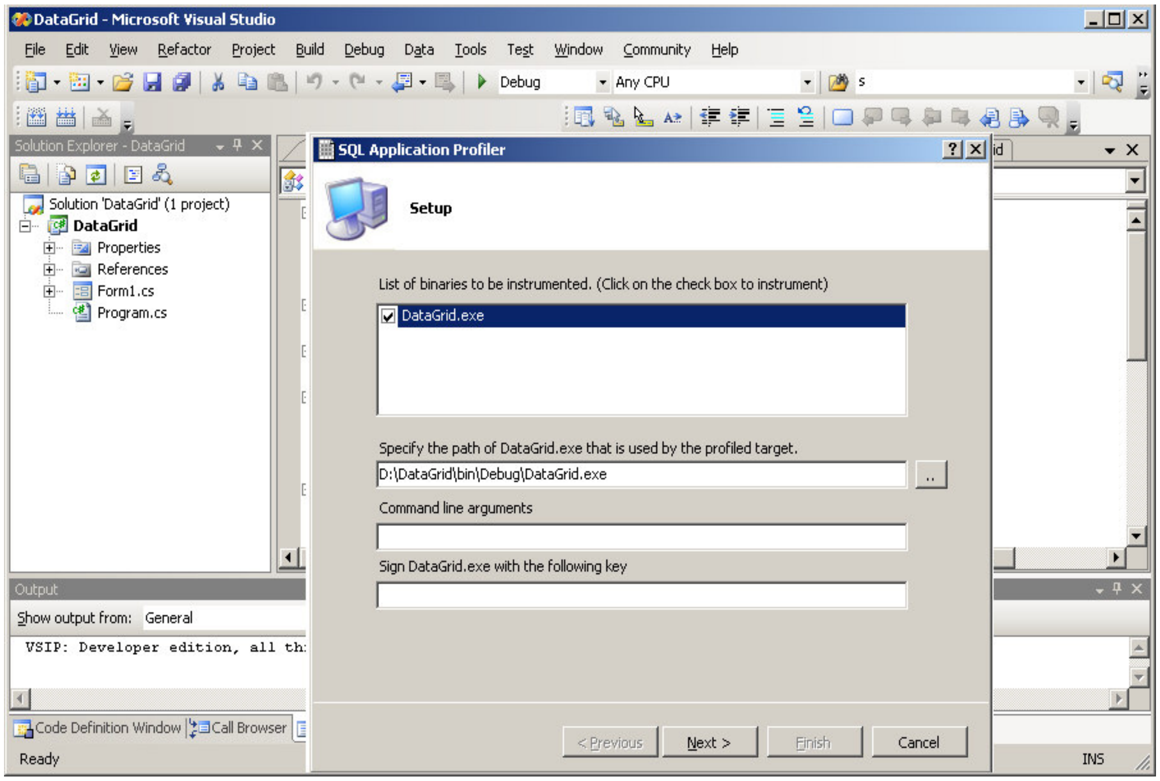
The architecture of our infrastructure and the browsing and analysis tools we have built using the infrastructure is shown in Figure 1.

We have extended Visual Studio System (VSS) using VSS extensibility API's [5] to integrate the functionality of the tool. Our tool takes an input the application binaries. A typical usage scenario of the tool is to open the project corresponding to the target that needs to be profiled. In Figure 2 the developer opened the DataGrid project and clicked on an Add-In menu of Visual Studio to invoke the tool. Since we have integrated the tool into VSS, it can automatically enumerate the VSS project system to get the target executable file(s) to be instrumented and determine any dependencies (like dynamically linked libraries) that may also need to be instrumented. Alternatively the developer has a choice to point the tool to a specific set of binaries. The tool enables developers to profile any .NET application that uses ADO.NET interfaces to talk to a Microsoft SQL Server 2005 database server.

Once instrumented, developer can click through the wizard, 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 that correlates application and server events using a set of matching techniques (see Section 4.3). 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 value to database developers.

Once the post-processing step is complete, the tool invokes the module corresponding to the summary/drill down box in Figure 1. The output of the tool is the summary/detail view as shown in the Figure 3 below. 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 functional overview and usage of the tool is described below.

The Summary view gives the function name, aggregate time spent in a function, how many times the function was invoked and aggregate time spent 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.


DataGrid - Microsoft Visual Studio
File Edit View Refactor Project Build Debug Data Tools Test Window Community Help
Debug
Any CPU
Solution Explorer - DataGrid
Solution 'DataGrid' (1 project)
DataGrid
Properties
References
Form1.cs
Program.cs
Output
Code Definition Window
Call Browser
Ready
INS
SQL Application Profiler
Setup
< Previous
Next >
Finish
Cancel

List of binaries to be instrumented. (Click on the check box to instrument)

[✓] DataGrid.exe

Specify the path of DataGrid.exe that is used by the profiled target.

D:\DataGrid\bin\Debug\DataGrid.exe

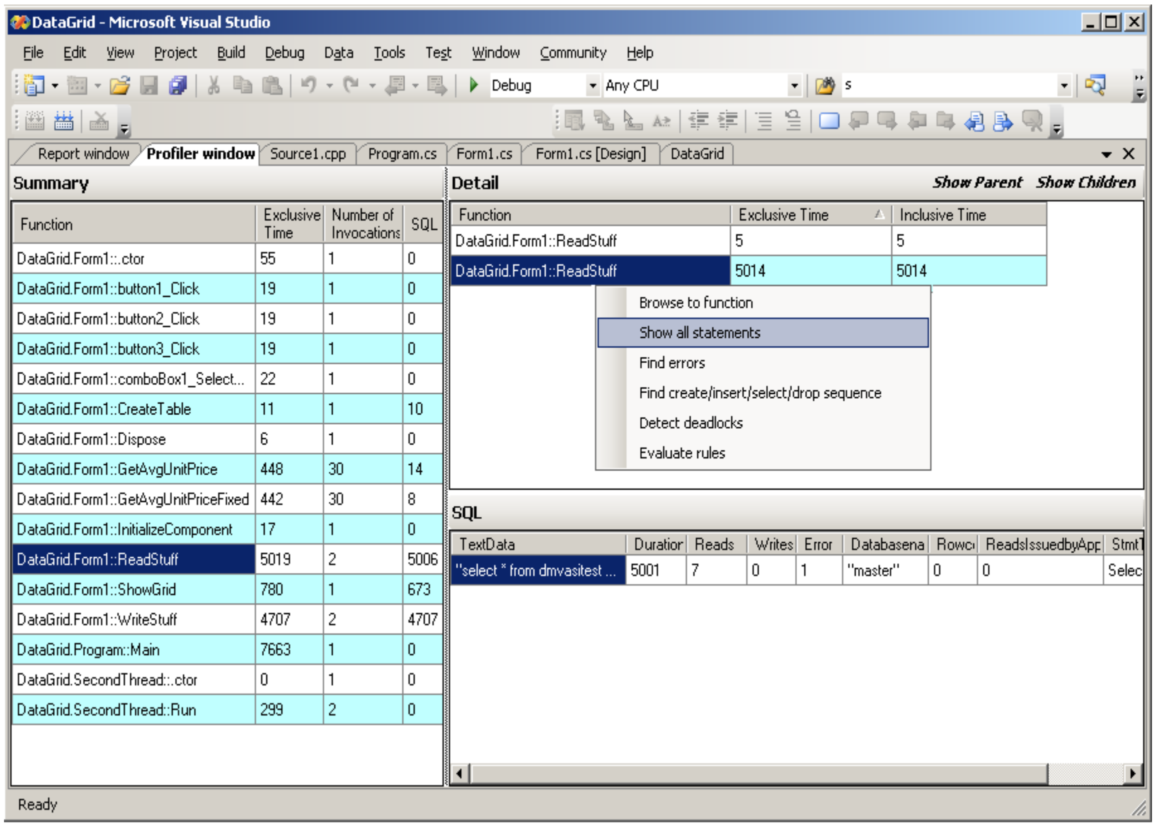
Command line arguments

Sign DataGrid.exe with the following key

Show output from: General

```
VSIP: Developer edition, all th
```

Figure 2. Instrumenting the target application.



Summary

Function	Exclusive Time	Number of Invocations	SQL
DataGrid.Form1::.ctor	55	1	0
DataGrid.Form1::button1_Click	19	1	0
DataGrid.Form1::button2_Click	19	1	0
DataGrid.Form1::button3_Click	19	1	0
DataGrid.Form1::comboBox1_Select...	22	1	0
DataGrid.Form1::CreateTable	11	1	10
DataGrid.Form1::Dispose	6	1	0
DataGrid.Form1::GetAvgUnitPrice	448	30	14
DataGrid.Form1::GetAvgUnitPriceFixed	442	30	8
DataGrid.Form1::InitializeComponent	17	1	0
DataGrid.Form1::ReadStuff	5019	2	5006
DataGrid.Form1::ShowGrid	780	1	673
DataGrid.Form1::WriteStuff	4707	2	4707
DataGrid.Program::Main	7663	1	0
DataGrid.SecondThread::.ctor	0	1	0
DataGrid.SecondThread::Run	299	2	0

Detail

Function	Exclusive Time	Inclusive Time
DataGrid.Form1::ReadStuff	5	5
DataGrid.Form1::ReadStuff	5014	5014

SQL

TextData	Duratior	Reads	Writes	Error	Databasena	Rowc	ReadsIssuedbyApp	StmtT
"select * from dmvasitest ...	5001	7	0	1	"master"	0	0	Selec

Figure 3. Summary and Detail views on the profiled data

Consider the function ReadStuff in Figure 3 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 ms. Out of the total time spend in the function, most of the time was spend executing SQL (5006 ms). 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.

Finally, the tool allows interesting analysis tools to be built on top of the profiled and correlated trace data such as suggesting FAST-k query hint (see Section 5.2), finding patterns of SQL statement invocations (see Section 2.5), detecting deadlocks (see Example 1), finding connection object leaks, etc.

4. IMPLEMENTATION

We now describe the implementation of our key components. As explained in Section 3, our tool takes as input the binaries of the application, which are then instrumented (Section 4.1). When the instrumented binaries are executed, they generate a log of events from the application as well as the database server (Section 4.2). These events are then analyzed by a log analyzer (Section 4.3) which produces as output a schematized view of the application trace. This view allows flexible analysis over the trace and allows tasks such as those described in Sections 1 and 2.

4.1 Binary Instrumentation Module

We use a binary instrumentation toolkit called Vulcan [10]. Vulcan works by analyzing the input binary and converting it into a symbolic intermediate representation (IR) which is easy to manipulate. Vulcan provides high level abstractions of its IR that allow a programmer to traverse the various components of a binary (Program, Component, Section, Procedure, BasicBlock, and Instruction) and to modify it. We examine the binaries and we insert instructions for instrumentation at selected places. Currently, we instrument function enter/exit. When the instrumented binary is executed and the program enters or exits a function it makes a callback into our module. This module generates a custom trace event which encapsulates attributes like the hostname, process id, thread id, time stamp and the name of the function. This event is then written to the current trace session. Events from this session are logged as described below in Section 4.2.

4.2 Event Providers and Logging

We use the Event Tracing for Windows (ETW) infrastructure which is a low overhead event logging mechanism used by a variety of Windows applications. An event represents any activity of interest and is customizable by the *event provider*. Every event logged to an ETW *session* contains some common attributes like the name of the event provider, type of event, id of the thread that issued the event, time stamp, and duration. In addition there is an attribute that allows provider defined data to be logged. For instance, as described above, events logged by the instrumented application describe the name of the function that was entered (or left).

We leverage the fact that Microsoft SQL Server 2005 and the data access layer (ADO.NET, ODBC, and OLEDB) is instrumented to emit ETW events on demand. The Microsoft SQL Server event provider can emit event types like login audits, stored procedure execution completed, batch execution completed, deadlock etc. Also the server event provider emits custom data that has various interesting attributes like duration, rows returned by server, number of reads and writes etc. The ADO.NET layer provides events corresponding to every data read API, opening and closing of connections, type of data access API used. When the instrumented application is run we use ETW's trace control manager interface to turn on emitting of events by the three providers: application's instrumented binary, data access layer and database server layer. We also ensure that all the events are logged to a unique session. Thus, the timestamps of events across processes on the same machine are generated using a single mechanism, which make correlating these events easier. The single log corresponding to this session can subsequently be analyzed offline after the application has finished running.

As an example, for the function shown in Figure 4, which uses ADO.NET APIs to connect to the database server, the trace log shown in Table 1 is generated. Note the three types of Events: Application, Data access and Server. The common information available across processes is timestamps. The provider data column contains the actual payload of the event (e.g., in the form of SQL statement strings etc).

4.3 Log Analyzer: Matching application and database context

The log analyzer takes as input the trace log. It correlates the events from the three providers and produces as output a schematized view of the application trace that contains both application and server context information. An example of this (represented in XML) is shown in Figure 5. Note how the output shows the thread id, function name, the SQL statement issued by the function, database server context for the statement. This output allows flexible analysis over the trace and allows tuning and debugging tasks such those described in Section 2. The key value added by the Log Analyzer is in correlating the application and database contexts from the event log.

```
Function foo() {
        SqlConnection conn = new SqlConnection;
        conn.Open();
        SqlCommand cmd = new SqlCommand(cmdtext,
        conn)
        cmdtext = "select * from T";
        cmd.ExecuteReader();
        conn.Close();
}
```

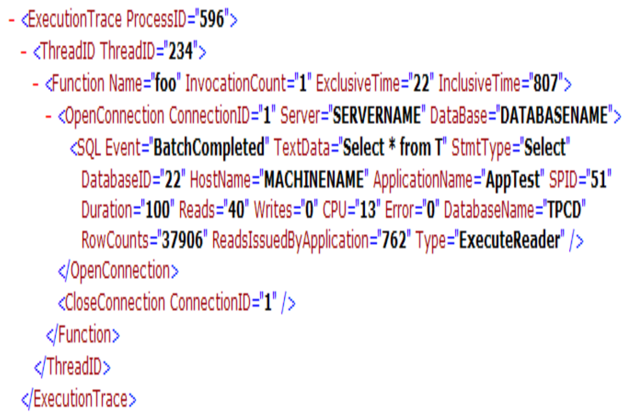
Figure 4. Sample application code that opens a connection, executes a query, and closes the connection.

Events have the schema shown in Table 1 (only a few relevant attributes are shown).

The log analyzer needs to correlate the three types of events: Application Events (function enter/leave), data access (ADO.NET) events and database server (Microsoft SQL Server 2005) events. Correlating an Application event and an ADO.NET event is relatively straightforward since both these kinds of events are emitted from the same process. In particular, given the ThreadId and Timestamp, it is possible to correlate exactly the data access events that are executed within a particular invocation of a function.

Table 1. Example of a trace log generated when the function shown in Figure 2 is executed.

Event Type	Proc. id	Thread id	Timestamp	Provider data
App Event	596	234	403065	"Enter {foo}"
Data access Event	596	234	403075	"Open connection"
Data access Event	596	234	403085	"GetCommand API"
Data access Event	596	234	403095	Set text "select * from T""
Data access Event	596	234	403105	"SqlCommand.ExecuteReader"
Server Event	128	104	403115	"select * FROM T"; Duration=100; Reads=40…"
Data access Event	596	234	403125	"Close connection"
App Event	596	234	40135	"Leave {foo}"



```
- <ExecutionTrace ProcessID="596">
 - <ThreadID ThreadID="234">
  - <Function Name="foo" InvocationCount="1" ExclusiveTime="22" InclusiveTime="807">
   - <OpenConnection ConnectionID="1" Server="SERVERNAME" DataBase="DATABASENAME">
      <SQL Event="BatchCompleted" TextData="Select * from T" StmtType="Select"
       DatabaseID="22" HostName="MACHINENAME" ApplicationName="AppTest" SPID="51"
       Duration="100" Reads="40" Writes="0" CPU="13" Error="0" DatabaseName="TPCD"
       RowCounts="37906" ReadsIssuedByApplication="762" Type="ExecuteReader" />
     </OpenConnection>
     <CloseConnection ConnectionID="1" />
    </Function>
   </ThreadID>
  </ExecutionTrace>
```

Figure 5. Example of output of Log Analyzer for input shown in Table 1.

Matching a Data Access event **D** with the corresponding Database server event(s) is more challenging. Ideally, we need an identifier for an event that is *unique* across the application and server processes. However, such a unique identifier is unavailable today for data access providers against Microsoft SQL Server. Until such support is made available, this matching remains challenging and is a key step where we add value.

One useful attribute available from each event in the database server side is the client process id. Thus a data access event **D** is matched with the server event **S** only if the D.ProcessId = S.ClientProcessId. However, since a single process may issues multiple SQL statements concurrently (e.g., on different threads), we need additional techniques for narrowing down the possible matches.

We have two additional pieces of information to assist matching. First is the timestamp. For example, we know that a data access event such as SqlCommand.Execute must *precede* the corresponding server event which is emitted once the SQL command has executed. This can significantly narrow down the possible server events that can match. Second, the provider data contains the actual string of the command being executed. The main issue with relying on matching strings is that exact string matching is not robust for this task. This is because the string by the data access layer may get modified in the server. Therefore, instead of exact string matches we rely on approximate matching. There are many techniques that have been developed for approximately matching string, e.g., [8]. We use a simplified version of [17] that requires tokenizing the strings based on delimiters and computing the intersection of tokens between the two strings. This technique is significantly more reliable than exact string matching, and has worked well in the applications on which we have used the tool thus far.

```
MatchEvents (S) // S is event stream obtained via ETW
tracing of App and SQL Server ordered by TimeStamp
1.  While (Events available in event stream)
2.    Parse the event stream to gather event type, thread
      id, user data (the SQL string)
3.    If the EventType is an App event
4.      Extract the SQL string that was issued, add to the
        current thread context. Add invoking function
        name.
5.    If the EventType is a SQL Server Event
6.      ServerString = Get the SQL string of the event
7.      For each thread context available
8.        For each event in the current thread context
9.          AppString = Get the SQL string of the event
10.         Score = GetApproximateMatchScore
                    (ServerString, AppString)
11.         If (Score > MaxScore)
12.           MaxScore = Score
              MatchedThreadContext =
                    CurrentThreadContext
13.       Output the AppString with the highest score as
    the match for ServerString and remove from the
    MatchedThreadContext
```

Figure 6. Algorithm for matching application events with database server events.

Our algorithm for matching used by the log analyzer is sketched in Figure 6. The algorithm maintains a *Thread context*, which is a data structure that maintains a list of strings corresponding to the SQL events issued by this thread that are not yet matched to a Microsoft SQL Server event. We have observed that in the real-world application that we have used the tool with, we get very good accuracy of matches using the above techniques.

Finally, note that the above algorithm is applicable even when the traces are generated on multiple machines (e.g., database server and application execute on different machines). The only difference is that the event stream S is now the "merged" trace across the machines, merged by the event timestamp field. It is necessary to take into account clock skew across machines while merging, for example by using the Network Time Protocol [23].

4.4 Techniques for Controlling Instrumentation and Tracing Overheads

Our tool's focus is currently for scenarios during application development and debugging, where instrumentation overheads are typically not an issue. To use the above technology for production setting of course requires a more careful examination of these overheads. Below we briefly mention a few techniques for controlling these overheads (for details of overheads of our current prototype see Section 6).

- **Static analysis to detect only relevant functions**. Static analysis of a binary can help determine if a function can possibly perform a data access call. We could then only instrument such functions. Note that with this optimization, since we do not instrument functions without data access calls, the call graph of function calls made by the application may be lost.
- **User specified modules, functions.** The developer may be aware of which binaries, or even which functions are of interest for profiling. Our binary instrumentation module (Section 4.1) can take a list of such binaries/functions as input and only instrument the specified functions.
- **Selectively turning on data access and database server events.** We can ensure that we request ADO.NET and Microsoft SQL Server the providers to emit only necessary events. For example, we can turn off events in ADO.NET that are emitted each time an application consumes a row. In this case, we trade-off instrumentation overhead for some application context information (number of rows consumed by application).

5. VERTICAL DEBUGGING AND TUNING TOOLS

As described in Section 4, the log analyzer produces a schematized view of the application trace that contains both application and server context information. This view generated can subsequently be queried for providing browse, summarize and drill down functionality. For example the tool allows developers to explore function call hierarchies and allows drill down on SQL statements that were issued under a function/call tree.

In this section, we describe two verticals we have built that perform more advanced analysis over the profiled data. The first example shows how to detect functions in the application that caused a deadlock in the DBMS (see Example 1 presented in the Introduction). The second example describes how to recommend to the application developer the suggested use of an OPTION (FAST k) query hint for an expensive query whose entire result set may not be consumed by the application (see Section 2.1).

5.1 Detecting Applications Functions causing a Server Deadlock

The Microsoft SQL Server 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 process id(s) information. The log analyzer (Section 4.3) extracts this information and stores it in the schematized application trace under the root node of the tree (as an event of type deadlock).

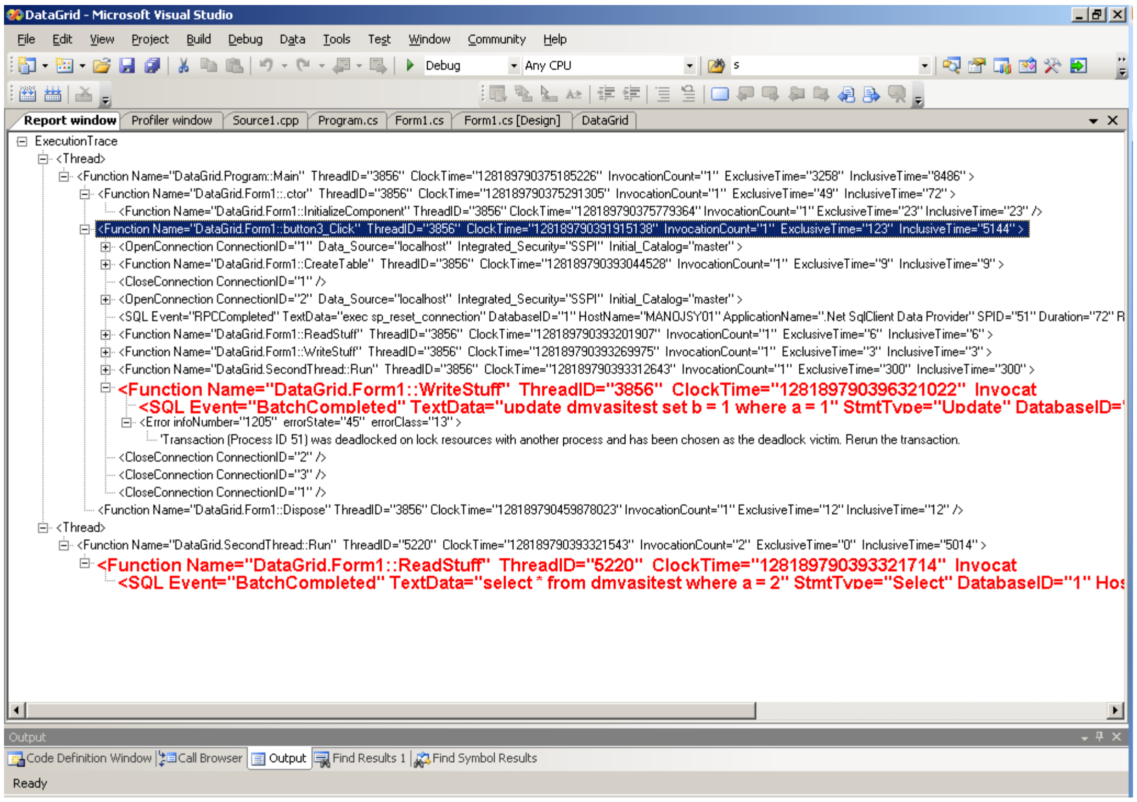
For each such deadlock event, the deadlock analysis "vertical" finds the statements issued by the application that correspond to the statements in the deadlock event. Note that once we find the statement, we get all its associated application context such as function and thread. This can then be highlighted to the developer so they can see exactly which functions in their application issues the statements that lead to the deadlock (as described in Example 1 in the Introduction). A sample output from the deadlock analysis vertical is shown in Figure 7.

The output of the log analyzer (described in Figure 5) is expanded to the right level and the functions and the SQL events that caused the deadlock in the server are highlighted using a color coding scheme.

5.2 Suggesting FAST-k query hints

In Section 2.1 we present the example of a query that returns many rows of which only a few are consumed by the application. We observe that in such cases significant speed up may be possible if the application developer can rewrite the query to pass in an OPTION (FAST k) query hint to the database server, so that the query optimizer can choose a plan that more optimal when k rows are needed (as opposed to all rows needed). Thus the developer can point to a query and invoke the *Fast-k analysis tool* which returns as output an analysis of how the cost of the query varies with k (see Figure 8 for an example of the output). Such information can be used by the developer to decide if it is appropriate to rewrite his/her query to use the hint.

The Fast-k analysis tool explores how the cost of the query varies with k (in OPTION (FAST k) hint). The naïve approach of costing the query for each value of k is not scalable. The key assumption we leverage is that in a well behaved query optimizer, the cost of the query plan cannot decrease as k is increased. For a large class of queries (such as single block SPJ queries with grouping/aggregation) this assumption typically holds true. Our approach is to perform a binary search over the range of values of k (between k_{min} and k_{max}), where k_{min} is the number of rows consumed by the application and k_{max} is the total number of rows returned by the query. Note that both these pieces of information are available from the output of the log analyzer. If the plan (and hence the cost) of the query remains the same for two different values of k (say k_1 and k_2), then we know that the plan (and cost) remains the same for all values of k between k_1 and k_2 as well. Thus, the binary search strategy allows us to prune out a large part of search space quickly.



```
ExecutionTrace
  <Thread>
    <Function Name="DataGrid.Program::Main"  ThreadID="3856"  ClockTime="128189790375185226"  InvocationCount="1"  ExclusiveTime="3258"  InclusiveTime="8486" >
      <Function Name="DataGrid.Form1::.ctor"  ThreadID="3856"  ClockTime="128189790375291305"  InvocationCount="1"  ExclusiveTime="49"  InclusiveTime="72" >
        <Function Name="DataGrid.Form1::InitializeComponent" ThreadID="3856" ClockTime="128189790375779364" InvocationCount="1" ExclusiveTime="23" InclusiveTime="23" />
      <Function Name="DataGrid.Form1::button3_Click"  ThreadID="3856"  ClockTime="128189790391915138"  InvocationCount="1"  ExclusiveTime="123"  InclusiveTime="5144" >
        <OpenConnection ConnectionID="1"  Data_Source="localhost"  Integrated_Security="SSPI"  Initial_Catalog="master" >
        <Function Name="DataGrid.Form1::CreateTable"  ThreadID="3856"  ClockTime="128189790393044528"  InvocationCount="1"  ExclusiveTime="9"  InclusiveTime="9" >
        <CloseConnection ConnectionID="1" />
        <OpenConnection ConnectionID="2"  Data_Source="localhost"  Integrated_Security="SSPI"  Initial_Catalog="master" >
        <SQL Event="RPCCompleted" TextData="exec sp_reset_connection" DatabaseID="1" HostName="MANOJSY01" ApplicationName=".Net SqlClient Data Provider" SPID="51" Duration="72" R
        <Function Name="DataGrid.Form1::ReadStuff"  ThreadID="3856"  ClockTime="128189790393201907"  InvocationCount="1"  ExclusiveTime="6"  InclusiveTime="6" >
        <Function Name="DataGrid.Form1::WriteStuff"  ThreadID="3856"  ClockTime="128189790393269975"  InvocationCount="1"  ExclusiveTime="3"  InclusiveTime="3" >
        <Function Name="DataGrid.SecondThread::Run"  ThreadID="3856"  ClockTime="128189790393312643"  InvocationCount="1"  ExclusiveTime="300"  InclusiveTime="300" >
        <Function Name="DataGrid.Form1::WriteStuff"  ThreadID="3856"  ClockTime="128189790396321022"  Invocat
          <SQL Event="BatchCompleted" TextData="update dmvasitest set b = 1 where a = 1" StmtType="Update" DatabaseID="
          <Error infoNumber="1205"  errorState="45"  errorClass="13" >
            'Transaction (Process ID 51) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.
        <CloseConnection ConnectionID="2" />
        <CloseConnection ConnectionID="3" />
        <CloseConnection ConnectionID="1" />
      <Function Name="DataGrid.Form1::Dispose" ThreadID="3856" ClockTime="128189790459878023" InvocationCount="1" ExclusiveTime="12" InclusiveTime="12" />
  <Thread>
    <Function Name="DataGrid.SecondThread::Run"  ThreadID="5220"  ClockTime="128189790393321543"  InvocationCount="2"  ExclusiveTime="0"  InclusiveTime="5014" >
      <Function Name="DataGrid.Form1::ReadStuff"  ThreadID="5220"  ClockTime="128189790393321714"  Invocat
        <SQL Event="BatchCompleted" TextData="select * from dmvasitest where a = 2" StmtType="Select" DatabaseID="1" Hos
```

Figure 7. Output of deadlock analysis vertical.

An example of the output produced by our tool is shown in Figure 8. In this figure Query Cost is the optimizer estimated cost of the query. Around the value of k=1000, the query optimizer switches the plan from an Index Nested Loops join to a Hash join. This causes the cost of the plan to increase significantly as shown. By observing such output, the developer could determine whether providing the OPTION (FAST k) query hint is appropriate for his/her application or not.

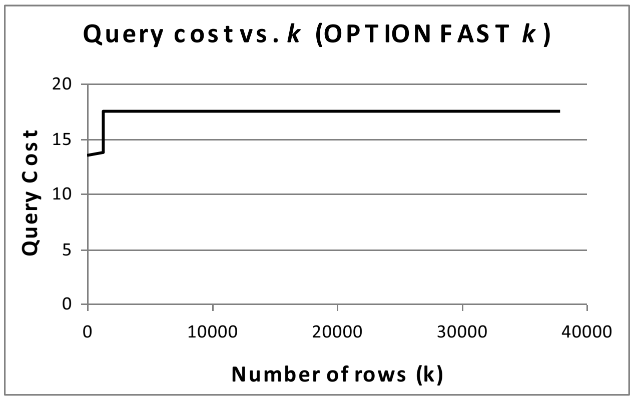


Figure 8. Analysis of query cost vs. *k* for the OPTION (FAST *k*) query hint.

6. EXPERIENCES WITH REAL APPLICATIONS

In this section we report evaluation of the tool over four applications:

- Microsoft SQL Server Database Engine Tuning Advisor (DTA)[2]. DTA is a physical database design tool that ships with Microsoft SQL Server 2005.
- Microsoft Conference Management Toolkit (CMT).CMT is a web service sponsored by Microsoft Research that handles workflow for an academic conference (http://msrcmt.research.microsoft.com/cmt/). CMT was traced for it "Paper Submission" functionality.
- MARS and LOB which are sample applications that run against the AdventureWork database that is shipped with Microsoft SQL Server 2005. LOB reads binary data from a file into a database and writes the contents to a file. MARS is a simple application that demonstrates the support for multiple access record sets in Microsoft SQL Server.

The summary of results is shown in Table 2. Note that while the instrumentation overheads are significant, these are typically not an issue when used by a developer in a development environment (see Section 4.4 for a discussion of optimizations for use in production scenarios). In one of the applications the profiling revealed that not all stored procedure invocations were invoked as RPCs, rather they were performed using the significantly less efficient Language events (see Section 2.4). In another application, running our tool revealed a set of redundant SQL statement invocations that was issued in the function that was responsible for opening connections.

Table 2. Summary of results of running the tool on four applications.

Application	No. of App events	No. of data trace events	No. of server trace events	Total events	Instrumentation overhead in the running time
MARS	25	87	15	127	7.1%
LOB	16	32	5	53	6.3%
CMT	6797	7224	515	14536	75.1%
DTA	3832951	3630	330	3836911	325.33%

7. RELATED WORK

Relational DBMS vendors provide profiler tools that are commonly used to profile database servers. Microsoft SQL Server Profiler [29], Oracle Trace [25] and DB2 Query Patroller [15] are widely used by database developers. Similarly database development environments such as Microsoft Visual Studio profiler [22], Rational PurifyPlus [27], gprof [13] etc. all provide application profiling capabilities. But as explained in the introduction section, to the best of our knowledge these two sets of tools remain disconnected from one another. Our infrastructure builds upon the profiling capabilities of both application side and server side profilers, and adds value by correlating context across server and application contexts.

There are integrated profilers like Mercury Diagnostics Profiler™ [19] and profilers from Identify [4]. While technical details are not available about specifics of these tools, they do provide some degree of correlation of application activity with database activity. Our work on building more advanced vertical tools such as those described in Section 5 is novel with respect to these tools. Magpie is a framework [6] that performs function level instrumentation of Microsoft SQL Server itself, and uses ETW to log its trace, but is not geared for gathering the context of SQL/ADO.NET applications as we do.

Finally, there are several tools that can help database developers tune their SQL such as Database Engine Tuning Advisor (DTA) [2], IBM DB2 Design Advisor [14], Oracle Tuning Pack [26], Quest [28]. However, the key difference is that these tools work exclusively on the database context, and do not leverage application context as described in this paper.

8. CONCLUSION

We have developed a tool that helps bridge the disconnect that exists today between application side profilers and DBMS profilers. Our tool allows database application developers to perform debugging and tuning tasks that are difficult or impossible using tools available today. An ongoing direction of work is developing more vertical analysis capabilities into our tool.

9. REFERENCES

[1] ADO.NET. http://msdn2.microsoft.com/en-us/library/aa286484.aspx

[2] Agrawal, S., Chaudhuri, S., Kollar, L., Marathe, A., Narasayya, V. and Syamala, M .Database tuning advisor for Microsoft SQL Server 2005. SIGMOD 2005

[3] Agrawal, S., Chu, E. and Narasayya, V. Automatic Physical Design Tuning: Workload as a Sequence. SIGMOD 2006.

[4] AppSight. Application Monitoring for Windows/.NET from Identify Software. http://www.identify.com/products/win-net/monitor.php

[5] Automation and Extensibility for Visual Studio. http://msdn2.microsoft.com/en-us/library/xc52cke4(VS.80).aspx

[6] Barham, P., Isaacs, R., Mortier, R. and Narayanan, D. Magpie: online modeling and performance-aware systems. *9th Workshop on Hot Topics in Operating Systems (HotOS IX)* (May 2003) 85-90

[7] Beauchemin, B., Perret, J., and Bunch, A. Data Access Tracing in SQL Server 2005. http://msdn2.microsoft.com/en-us/library/aa964124.aspx

[8] Chaudhuri, S., Ganjam, K., Ganti, V. and Motwani, R. Robust and efficient fuzzy match for online data cleaning. SIGMOD 2003, 313-324

[9] Eclipse Test and Performance Tools Platform. http://www.eclipse.org/tptp/index.html

[10] Edwards, A., Srivastava, A., and Vo, H. Vulcan. Binary transformation in a distributed environment. *MSR-TR-2001-50* (April 2001).

[11] Event Tracing. http://msdn2.microsoft.com/en-us/library/aa363787.aspx

[12] ExecuteScalar API. http://msdn2.microsoft.com/en-us/library/system.data.sqlclient.sqlcommand.executescalar.aspx

[13] Gprof. http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_mono/gprof.html

[14] IBM DB2 Design Advisor. http://www.ibm.com

[15] IBM DB2 Query Patroller. http://www-306.ibm.com/software/data/db2/querypatroller/

[16] Java Database Connectivity. http://java.sun.com/javase/6/docs/technotes/guides/jdbc/

[17] Kirpal, A. and Sarawagi, S. Efficient set joins on similarity predicates. SIGMOD 2004, 743-754

[18] Meier, J.D, Vasireddy, S., Babbar, A. and Mackman, A. Improving ADO.NET Performance. *Patterns & Practices* (May 2004), Chapter 12. http://msdn2.microsoft.com/en-us/library/ms998569.aspx

[19] Mercury Diagnostics, J2EE Performance, SAP Diagnostics, .NET, ERP/CRM Diagnostics. http://www.mercury.com/us/products/diagnostics/

[20] Microsoft Visual Studio 2005. http://msdn2.microsoft.com/en-us/vstudio/default.aspx

[21] Microsoft® Open Database Connectivity (ODBC) interface. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/odbc/htm/dasdkodbcoverview.asp.

[22] Microsoft Visual Studio profiler. http://msdn2.microsoft.com/en-us/library/z9z62c29(VS.80).aspx

[23] Network Time Protocol. http://www.ntp.org.

[24] OLE DB. http://msdn2.microsoft.com/en-us/data/default.aspx

[25] Oracle Trace. http://www.oracle.com

[26] Oracle Tuning Pack. http://www.oracle.com/technology/products/oem/files/tp.html

[27] Rational PurifyPlus. http://www-306.ibm.com/software/awdtools/purifyplus/

[28] SQL Server database tools and management software from Quest Software. http://www.quest.com/sql_server/

[29] SQL Server Profiler Reference. http://msdn2.microsoft.com/en-us/library/ms173757.aspx

[30] TPC Benchmark H. Decision Support. http://www.tpc.org.