Aspy
An Access-Logging Tool for JDBC Applications

A. Torrentí-Román, L. Pascual-Miret, L. Irún-Briz, S. Beyer, F. D. Muñoz-Escoí

Instituto Tecnológico de Informática
Ciudad Politécnica de la Innovación, 8G
C. Ing. Fausto Elio, s/n
46022 Valencia, Spain

{alx,let,lirun,stefan,fmunyoz}@iti.upv.es

Technical Report TR-ITI-ITE-07/24

Aspy An Access-Logging Tool for JDBC Applications

A. Torrentí-Román, L. Pascual-Miret, L. Irún-Briz, S. Beyer, F. D. Muñoz-Escoí

Instituto Tecnológico de Informática
Ciudad Politécnica de la Innovación, 8G
C. Ing. Fausto Elio, s/n
46022 Valencia, Spain

Technical Report TR-ITI-ITE-07/24

e-mail: {alx,let,lirun,stefan,fmunyoz}@iti.upv.es

Dec, 2007

Abstract

When different developer teams collaborate in the design and implementation of a large –and distributed– application, some care should be taken regarding the access to persistent data, since different components might use their own transactions and they might collide quite often, generating undesired blocking intervals. Additionally, when third-party libraries are used, they can provide unclear descriptions of their functionality and programmers might mistakingly use some of their operations. An access logger can be useful in both cases, registering the sentences actually sent to the database and the results of said sentences. *Aspy* is a tool of this kind, developed as a JDBC-driver wrapper for Java applications. It is able to save in a file the list of calls received by the JDBC driver, registering their parameters, starting time, completion time and either their obtained results or their raised exceptions. With such information, it is easy to identify common errors in database accesses and the set of transactions involved in blocking situations due to poor application design. We discuss three different techniques that were used for implementing *Aspy*, comparing their pros and cons.

1 Introduction

Client-server distributed applications are commonly built following a three-layer architecture. The bottom layer is regularly implemented using a Database management system (DBMS), on top of which a business logic layer is developed. When a large client-server application is designed, multiple components may be identified in such an intermediate layer and each of them may be developed by a different group of programmers. Debugging such applications might be difficult, especially when database accesses are delegated to third-party libraries/components. In such cases, a tool which is able to record the actual sentences being sent to the DBMS at run-time in a log, together with the execution time of each sentence, the connections where they initiated, and their results, could be highly interesting. Many bugs and program misbehaviors could be diagnosed using such tool.

To our knowledge, although some tools of this kind might exist, none of them are able to work independently of the underlying DBMS being used, nor in a completely transparent way. We propose *Aspy*, a JDBC-driver wrapper that is able to work with all DBMSs that provide a JDBC driver and that does not require any modifications to the user-level application in order to use it. Thus, the tool can be used by any Java application that accesses a DBMS using the JDBC interface, without requiring its recompilation nor any special deployment action.

Since the functionality of this access logger can be implemented using different approaches, we have selected three of them in order to have some initial prototypes. This work compares such approaches and gives an initial performance comparison of those that appeared better at design time.

The rest of this paper is structured as follows. Section 2 describes the objetives and requirements of the access-logging tool. Section 3 outlines the architectural approaches that might be followed for building the tool, discussing their pros and cons. Later, Section 4 describes the resulting tool, and Section 5 provides the results of an initial performance comparison. Finally, Section 6 outlines the applicability of this logger, Section 7 discusses further work, and Section 8 gives the paper conclusions.

2 Requirements

As already commented above, the aim of our tool is to log each one of the database accesses made by a given application. Such logging is interesting for diagnosing problems in the way a programmer has structured each one of the application transactions, or in how a set of concurrent transactions are involved in livelocks or deadlocks. If such situations can be analyzed afterwards, reading the appropriate logs, the programmer will be able to rewrite his/her program in order to fix the problems that have been detected. To this end, different issues should be required from this proposed tool: stable logging, efficiency, adaptability, transparency and appropriate error/exception reporting and management.

Stable logging. All database accesses should be logged in stable storage in order to allow an off-line analysis of the program behavior. Although on-line processing could also be made with some tools, and is also available in our tool, some types of log processing require a non-negligible time and logging in secondary storage is needed in such cases.

Efficiency. The logging process should be as fast as possible, in order to introduce a minimal delay in the application being monitored. Although we intend to use our tool in the development/testing/debugging phases of a target application, it is not discardable to use also the tool in already deployed applications.

Adaptability is needed because software components evolve in a continuous way. New compiler versions or database versions can be raised in question of months. If we can implement a logger that works for every software application and for any DBMS, then adaptation cost will be null and comparing those systems will be easy.

Transparency allows to replace the real driver with our tool without affecting the normal behavior of the application being monitored. This is required when we do not have enough resources to modify the application to include the desired functionality. Thus, ease of setup will ease the integration process. Transparency may even be required, if we cannot modify any source or the execution lines.

In the real world, a non-recommended way to program is to hide the exceptions raised by other application components, without appropriately managing them. The main objective of this incorrect practice is to provide a view of a non-faulty program but sometimes exceptions are non-predictable and they have been caused by unexpected events, so they should be reported. One example of this behavior is an application that ignores such exceptions, leading to a transaction rollback due to a constraint violation. Final user can think that the transaction was successfully committed –in case of exceptions not being reported–, but some operations caused a problem which should to be shown. A way to handle this situation is needed, which means that exceptions should be reported together with the problem that caused them. The way to introduce this new functionality should be transparent or at least leave the logic as simple as possible. Our aim is to be able to report those exceptions that were initially ignored by the user-level application or library that directly accessed the database.

In the next section, we explain how to reach these requirements comparing three different logging methods and showing their advantages and disadvantages.

3 Logging Techniques

In this section we make a qualitative comparison between three logging techniques: extending the connector interface, usage of dynamic proxies, and Aspect Oriented Programming (AOP).

3.1 Extending the connector interface

The first technique that is studied might be the simplest one. We reach the main goal, by implementing the main interfaces of the connector in different classes with pointers to an instance of the real driver. Then, we use the such extended driver instead of the one provided by the vendor. Every method that wants to be logged has to be rewritten, which implies a big effort when the connector API is complex. Later, at every rewritten method an invocation to the real driver is needed, to leave the system unaltered. Such behavior is outlined in the following pseudocode example:

```
void method(type value) {
    take initial time
    realDriver.method(value)
    take final time
    log(method, parameters,
      final time, initial time, ...)
}
```

This technique is useful when we need to log a few methods or do some specific operations for each method, otherwise its cost is excessive and all the methods of an interface have to be wrapped even if we are only interested in a few. To accomplish the goal of catching exceptions, code has to be surrounded with try-catch blocks. Logging the method and the value of the parameters that caused the exception is interesting to reflect the problem in order to offer the best description to the administrator. This practice is considered necessary and unavoidable when the software is still in its debug phase. The overload introduced by executing all the code of the wrapper between try-catch blocks can be suppressed at deployment phase if the software has been sufficiently tested.

To sum up, the main advantage of this approach is that we can use a different logging functionality in each one of the operations being monitored, since the wrapping code needs be explicitly written for all operations. On the other hand, its main problem consists in the need of wrapping each one of the operations being provided in the wrapped interface, demanding some programming effort even for operations that were of no interest.

3.2 Dynamic proxies

The weakness of the previous technique is corrected with reflection. The main advantage of this technique is that Java and other languages have an introspective property, thus, the state at execution time can be retrieved. If we combine this with a generical point of access (for instance, when an invocation takes place) then we can access to all methods with one small piece of code. One way to do that is using the dynamic proxy pattern, that wraps an instance of the real driver. Then, the user is provided with a set of proxies that implements the connector API. When an invocation is received, the proxy will pass the operation to the real driver and log the required information. Let's simplify and show the pseudo-code of a dynamic proxy:

```
RealDriver target

Object invoke(method, arguments) {
     take initial time
     method.invoke(target, arguments)
     take final time
     log(method, parameters,
       final time, initial time, ...)
}
```

Note that in this case we are extending each operation with code similar to that shown in Section 3.1 but now we are overloading a basic operation of a basic class, and thanks to the reflective characteristics of the Java language, such extensions are applied to each one of the operations in the RealDriver class (in this example).

As we can see, this way of logging guarantees a generic solution; i.e., we simply need to write a common code that will be used in all operations being filtered. As a result, this demands less programming effort. On the other hand, this may raise problems in those operations where we need specific behavior. The program logic might be more complex than with the previous technique –note that reflection needs some supporting code that needs to be included in all wrapped operations– and, as a result, there is a loss of performance. When log time arrives, most of the values need to be resolved at execution time, introducing an overload that has to be considered when high-performance systems are being logged. So for some cases, a method that introduces less impact should be used, such as a minimal wrapper constructed with the technique described in the previous section.

3.3 Aspect Oriented Programming

The Aspect Object Programming (AOP) is an intuitive technique to identify parts of the software where various concepts are being mixed, separating them into different aspects that may be developed independently. In Java there is an AOP framework called AspectJ. One of the most important characteristics of AspectJ is the fact that with aspects, new code –i.e., new functionality aspects– can be added to a compiled class or a bunch of classes, without needing their source code. Therefore, the new functionality can be inserted into the bytecode, only knowing which are the methods of the files, which is easy when we are using a public API with available interfaces. So, assuming that we have a basic knowledge of JDBC, we can generate new classes with the needed functionality and that will lead us to the main goal: to wrap JDBC. The way of doing that is setting some Pointcuts and define which operations will be added when the compiler finds such pointcuts. When the AspectJ compiler detects a method that matches a pointcut it adds new lines to the code. It is simple to make a rule which includes all the methods that implement any interface included in the JDBC API. Here is an example of this process in pseudo-code:

```
around(): execution(* * java.sql.*){
     take initial time
     proceed()
     take final time
     log(method, parameters,
       final time, initial time, ...)
}
```

This code is translated by the compiler. When an execution of a method occurs, then we measure time, proceed with the execution, and log the relevant information of the method, the time taken and any other relevant system parameters. When exceptions need to be caught, AOP is also a valid technique due its generality [8]. A pointcut can be added to offer this functionality by specifying that when an exception of the type thrown by a connector interface occurs, a log of the method and parameters that caused this exception and the cause, is needed. Obviously, this introduces an overload, as in the first wrapping technique, but two versions of the driver can be easily made by suppressing the exception pointcut. Thus, when the system needs a high performance, the driver without catching exceptions should be used. However, another version with exception logging can be easily made by setting the appropriate pointcuts. Anyway, logging exceptions is the best practice to determine which is the real behavior of the application, so administrators should take them into account.

4 An Example Tool: Aspy

Aspy is the fruit of this study. It is based on AOP and developed with AspectJ. Further optimization was done reflecting the study of [1]. AOP has been chosen for its benefits. On the one hand, we have its generality which allows to log every method included in the JDBC API. On the other hand, the main aspect can be applied to any particular JDBC Driver of any DBMS, so specific functionality can also be included. The gain is obvious, we can maintain a generic Driver in an easy way and we can develop a concrete Driver if we want to add to our DBMS specific features to reach a project requirement. We talked about different requirements that should be reached by our software in Section 2. Let's see how Aspy reaches them. *Stable logging*: Aspy is a piece of software located between the application and the DBMS. It logs into files all the relevant information of this interaction, like method and class names, parameters of the method, current time, an identifier of the connection relative to the object and the cost associated to the operation. Most of the tests will find this information enough to reach a satisfactory conclusion. *Efficiency*: AOP offers an agreement between generality and efficiency and the code can be optimized to the desired level with this technique. So, as we will see in Section 5, its main cost is derived from I/O access, so we have focused our efforts in data manipulation and in I/O management. *Adaptability*: AspectJ allows compiling the same aspect to different drivers. The only requirement on the driver's part is implementing the JDBC API, so the resulting code will be generated for all required methods. When the JDK changes its API, the aspect will remain the same, because these operations are not dependent on a version or a concrete API. *Transparency*: Aspy replaces the real driver, but all the functionality is included in the code. So, in a practical way, is like adding the log operations into the original source code. There is no need of specifying the real driver or to register both drivers, because Aspy is an extended version of the original driver. *Exception handling*: One difference between Aspy and p6spy, see Chapter 16 of [10], which is a software implemented using the first technique described in this paper, is exception reporting. As it was mentioned above, including this behavior is like duplicating the code, and AspectJ offers a simple way to log exceptions. The way to deal with this in AspectJ is as follows:. A pointcut is introduced in the aspect, which inserts code when a method throws a SQLException, so it will be inserted and caught at most methods that implement the JDBC API. The compiler will surround the methods that match with the previous description with a try-catch block, generating extra code. So a way to enable/disable exception logging is provided in order to minimize the impact due to reporting unexpected behavior. Enabling it is specially useful if an application runs without reporting the throwing of exceptions to the user, which is considered bad practice but often done.

4.1 Possible Configurations

Aspy can be configured to log only a few classes or a few methods or not to log some specific classes or methods. It is up to the administrator to choose which are the relevant classes and methods of the systems and which are not. Also, I/O parameters can be set up, for instance, to use specific appenders to log or to choose the way to receive the information.

4.2 Long-term program executions

Aspy was initially conceived to work while applications are in a debugging phase. However, there are many applications of the software that need the result log to be taken while the application is running and stable for months. Thus, the impact has to be minimum and its results should be organized and optimized. To deal with that, Aspy can be configured to generate multiple log files per days or per size. This eases the task of analyzing the logs for the administrator, which can make a periodical report of the performance of the application.

5 Aspy overload

To measure the overload introduced by Aspy in the system let's compare which are the operations made by it in a simple program. After that, we will see which is the overload due to Aspy logic to determine which are the methods to be logged and the data manipulation cost. Finally we will compare Aspy with p6spy and analyze the results. We do not consider a software made using reflection, as we consider reflection as a subset of AOP, because AOP uses all reflective functionality, improving it with extra tools to obtain more flexibility and usability.

When an invocation of a method is made, Aspy retrieves its parameters, measures the time before and after the operation and obtains the connection to which it belongs and the current time of the system. After appending each variable, it proceeds to write in the file or print to the console.

The following pseudocode could be an example for that situation:

```
try {
   obtain initial time
   execute operation
   obtain final time
   obtain return value
   obtain connection id associated
   write in file (currentTime,
   final - initial time, "className",
   "methodName", parameters +
   return value, connectionId)
}
catch (SQL Exception){
   write in file (currentTime,
   final - initial time, "className",
   "methodName", parameters,
   Exception launched, connectionId)
}
```

Equivalent Aspy pseudocode:

```
try {
obtain Signature
obtain methodName from Signature
obtain className from Signature
if (hasToBeLogged) {
  obtain initial time
  proceed with execution
  obtain final time
  obtain parameters from Signature
  obtain connection id from hashMap
  write in file (currentTime,
  final - initial time, "className",
  methodName,  connectionId
  parameters + return value)
}
}
catch (SQL Exception){
  write in file (currentTime,
  final - initial time,
  className, methodName, parameters,
  exception launched, connectionId)
}
```

Variables between ”” are obtained at compilation time, so their cost is lower than obtaining them at execution time.

The cost of a million of operations is:

```
Executing Example code:
145.577 seconds

With Aspy:
201.594 seconds
```

That means that most of the cost of logging is due to I/O operations and to reflection methods and not to adding a specific logic in order to determine which are the methods that should or should not be logged. Therefore, if the optimization is made by accumulating log lines and writing them by blocks will obtain a significant performance improvement.

All the tests have been made with exception advice enabled so an extra overload is introduced, but exceptions usually offer more information than statement logs, so their relevance demands them to be reported.

First test made with Aspy consists of a set of instructions simulating a high system load. This example is equivalent to a system based on a three-layer architecture, where the bottom layer is the Database layer, so, all the operations are JDBC operations. Considering a 100% of CPU use in a server with an Intel(R)

Pentium(R) 4 CPU 3.00GHz processor and 1024 KB of RAM and a PostgreSQL [4] Database, when the number of operations are increased progressively the overload is shown at Figure 1. X axis is the number of JDBC operations done and Y axis represents the time in seconds to execute these operations.

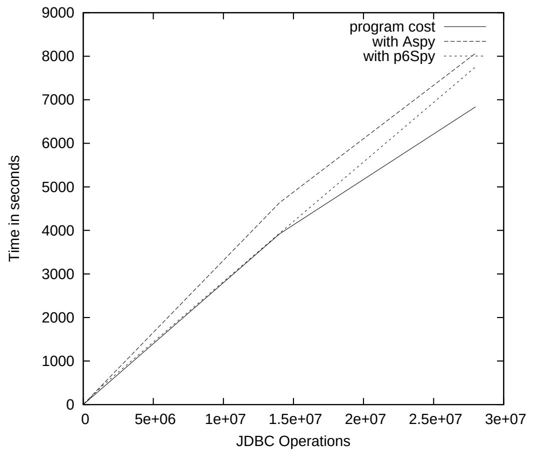

Figure 1: Aspy logging overload (intensive load).

For instance, for 280,000 JDBC operations, the application needs 74.2 seconds and the cost of using Aspy is 14.1 seconds. That means a 19% of overload.

A comparison with p6spy is also included, to justify the results. As we can see, initial overload is due to program logic. One of the main lacks of generality induced by Aspy is the use of reflection to retrieve the log information. When the number of operations is relative large, the impact introduced by Aspy is lower per operation, due to buffering optimizations. So for high-load applications where the use of CPU is also very high, logging is an overload that should be considered if the system needs a very high performance, and a 10% is relevant for the whole system. If the high loads are non-continuous, the first technique for logging is the best choice. The significant overload is introduced when the load is equal to 1,500,000 operations. Buffering techniques are needed after this threshold, to achieve a good performance ratio. For the AOP technique, a mixed approach has to be used. One way to gain performance is by eliminating execution time resolution of log values. Then a specific method for every JDBC operation is needed. We lose generality in benefit of performance. This can be done by generating logging code in an automatic way for every method and recompiling the driver. So, all the calls used to obtain pointcut state and its value are suppressed, generating a code equivalent to that shown in Section 3.1.

The second test is made by emulating a low-medium load in an application or server. It consists in doing a JDBC operation every 50ms in the same server as in the previous test. This system is equivalent to a user application connected to a database or a server with a normal transaction load. Results will show that overload in these systems is negligible, specially when buffering techniques are being used. Results are shown at Figure 2. X axis represents the number of JDBC operations and the Y axis the impact of the driver, expressed in percentage.

So, for non-high loads, logging is an interesting technique in every case, allowing to retrieve all the information generated by the system, so it can be used to improve it and to detect possible bugs or consistency errors. In next section we will describe the possible usages of logging in this kind of systems.

5.1 Buffering the log information

If Aspy is configured to write when the buffer reaches 100 lines, the overhead is 12% for high-workload applications, i.e., one-third lower than that shown in the configuration without using buffers, allowing thus continuous logging in a real world systems. This means that this technique is useful, especially if the factor is configured relative to the application where the logger is being used. One of the troubles that a log developer can find is knowing when the application is going to finish the execution. One way to deal with

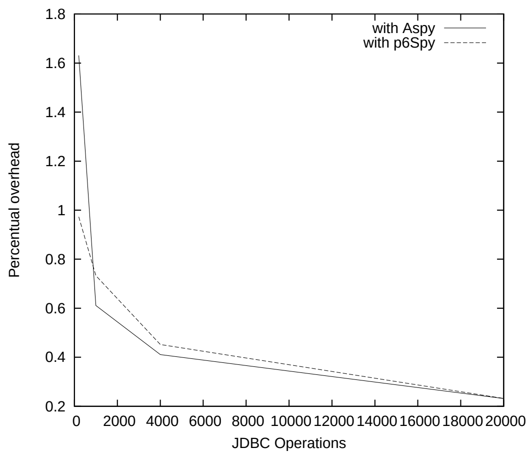

Figure 2: Aspy logging overload (medium load).

this situation is setting a thread hook, that runs when the main method ends, and flushes the buffer. So it is up to the administrator to achieve a good ratio of number of items in the buffer, to obtain the maximum performance.

So, the overload introduced by an operation logger should be considered by the final user or administrator in order to determine which is the level of log needed, because certain types of operations are not always required or they are not significant for some specific tests.

6 Applicability

As we introduced in earlier sections, knowing which are the real statements between a software application and the DBMS is an interesting tool. It leads us into a wide world of possible usages, which are very interesting and can be applied in different phases of software development. Furthermore, logging at this level makes the result independent of the DBMS, increasing the possibility to design common tests. In other works, such as [5], queries are analyzed at compilation time. In some cases, this is insufficient because most of the relevant parameters are visible at execution time, where Aspy can be used. As we discussed in Section 5, Aspy can replace the real driver without harming the performance, so real-time applications can be monitored even after the debug phase. Some of the further described applications of Aspy are being developed and some will be interesting to develop in the future, but this set of usages will increase the quality of the systems which use a Database, increasing their security and their performance.

1. Aspy can be used to detect incorrect configurations of a Database. For instance, a common mistake is not to deactivate auto-commit when we want to run a transaction. Then the application is committing every correct sentence, generating an incorrect state of the Database. Some DBMS raise an exception when this happens but others, like PostgreSQL [4] do not detect this situation, and allow the change of the isolation level without setting up to false the auto-commit flag. With Aspy we can deal with this situation and a modification of the code to avoid it is minimal, just by adding one line to the Driver when an Isolation Level change occurs.

2. Suspicious accesses to restricted tables can be detected when the code of the application is not provided, see [6] for more information. This is a very important utility for users and enterprises that have bought a third-party application/library and want to be sure about the confidentiality of their data. As in the first case, a simple modification can be made to directly avoid these operations or allowing only a subset of permitted ones.

3. Aspy generated logs can be the fed to an expert system that makes suggestions on query optimization to the programmer or on syntactic rules to the DBMS administrator. Our efforts are going in this way,

by analyzing performance techniques shown in *SQL Performance Tuning* to build optimizers which can detect and suggest changes to improve productivity. Note, however, that modern DBMSs have very powerful query optimizers and that our intent is simply to discourage the use of some sentences that can not be adequately optimized by the underlying DBMS.

4. Another kind of expert system, that might use Aspy, can help the administrator to improve the scheme, by suggesting indexes on the most accessed fields and tables. Due to the fact that all the information can be logged, this system can be made by identifying each field and table that was referenced in the operation and generating statistics with that information.

5. Cold replication can be made using the log file and a program that reproduces the operations, to another database. This is specially useful when heterogeneous systems, like different DBMS, want to be compared. It is also needed for raising DBMS at debug phase in order to detect implementation errors. It can be a fast alternative to [7].

6. In order to detect bad practices, such as ignoring exceptions by a deployed program, a logger with exception reporting can be used. So final users will detect forbidden accesses, failed connections, rollback operations and other behaviors produced when a communication with a database takes place. As it was mentioned before, this introduces an overload, so the decision should be taken by the administrator in order to maintain a good performance.

7 Related and Further Work

As mentioned earlier in this paper, p6spy is a good approach to the main goal, but its installation requires a more intrusive procedure and updating the whole tool represents an expensive cost of development. Some similar tools, like LOG4PLSQL [11], are also interesting but, when using LOG4PLSQL, the deployment is a hard process due to modifications that should take place at DBMS side. When comparing with Oracle Log Buffers, see [9], we can conclude that obtaining this information at DBMS level is a hard and expert process, and a specific knowledge of the DBMS and the version is required (to access to the committed and uncommitted operations). However, one main lack appears when using LOG4PLSQL and Oracle buffers, due to obtaining operation information at this level is less expressive than at top level, therefore, some relevant data will be lost (such as JDBC methods, parameters names, and their values) and it can be insufficient to feed the developer at the debug phase of the application.

Our efforts are focused on developing solutions that use the log generated by Aspy to reach the applications presented in Section 6. Some need a bigger effort than others, but most of them have a common approach, which can be implemented as a log line analyzer. Depending on every case, analysis can be done off-line or on-line but its goal is to provide advices to improve software quality and correctness.

We are planning to employ Aspy as the central component of a larger system, which provides support for database query analyzer modules. Analyzer modules can be either static or dynamic. Static modules process the Aspy log output offline, whereas dynamic modules are employed at run-time.

In order to support simple programming of analyzer modules, we intend to define a query language providing basic primitives for log processing. Furthermore, we will investigate the possibility of providing a graphic user interface that allows the construction of analyzers.

Finally, we plan to define an API that allows the interception and filtering of database accesses at run time. Such an interface could be used, for example, for security purposes.

8 Conclusion

Monitoring at this level, between the application and the DBMS, is a well-known practice. It is less intrusive than logging the whole program and provides enough information for debugging and optimization purposes, depending on the software phase in which the logging takes place. Impact minimization efforts are a justified action when long-term logging is needed. The use of AspectJ provides generic use for each DBMS and changes to the code are minimal if a special functionality is required. This property of Aspy is not incompatible with efficiency as we have shown in this paper. Considering the goals of our study, Aspy is a practical tool for logging and controlling the operations made by an application, especially when we do not have its source code, as it is the case with most third-party software. In other cases, where debugging is needed or a persistence layer is used, such as Hibernate [2] or [3], precise information of the operations is required, so with this kind of software the programmer can obtain a precise view of the real interaction and act in consequence; for instance, by creating new indexes over the database fields which are often accessed. So, after a first phase of analysis of the system requirements, a logger tool can be used to improve the software quality, in order to build better applications.

Acknowledgments

This work has been partially funded by EU FEDER, MEC and IMPIVA under grants TIN2006-14738-C02 and IMIDIC/2007/68.

References

[1] Pavel Avgustinov et al. Optimising AspectJ. *SIGPLAN Not.*, 40(6):117–128, 2005.

[2] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, CT, USA, 2006.

[3] R. Bloom. Debugging JDBC with a logging driver. *Java Developer's Journal*, 2006.

[4] Korry Douglas and Susan Douglas. *PostgreSQL*. New Riders Publishing, Thousand Oaks, CA, USA, 2003.

[5] Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. In *ICSE '04*, pages 645–654, Washington, DC, USA, 2004. IEEE-CS Press.

[6] Yi Hu and Brajendra Panda. Identification of malicious transactions in database systems. In *IDEAS03, Hong Kong, SAR*, pages 329–335. IEEE, jul 2003.

[7] Ganapathy Krishnamoorthy. Heterogeneous query processing through SQL table functions. In *ICDE 1999*, page 366, Washington, DC, USA, 1999. IEEE-CS Press.

[8] Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *ICSE '00*, pages 418–427, New York, NY, USA, 2000. ACM.

[9] Oracle Backup and Recovery. Oracle online documentation, 2008. URL: http://www.oracle.com.

[10] Jack Shirazi. *Java: Performance Tuning*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[11] The LOG4PLSQL project team. Log 4 pl/sql, 2002. Accessible in URL: http://log4plsql.sourceforge.net/.