

# HELP DEVELOPERS (FINALLY) FIND THEIR OWN PERFORMANCE DEFECTS

by Erik Ostermueller  
FIS Global

*How early in the software development cycle are most performance defects found? Before or after QA? Industry pundits have long sought to reduce costs by fixing software defects earlier in the cycle. The path to these cost reductions, however, is fraught with road blocks. This paper focuses on a concrete testing regimen that works around these longstanding obstacles. It empowers developers to finally help locate their own performance defects, instead of relying solely on the assistance of specialized performance tuning experts.*

## Motivations

This paper details a strategy for locating performance defects earlier in the development cycle. The assumption, trumpeted by testing advocates for decades, is that finding defects earlier in the cycle significantly reduces development costs. Some estimates claim that better testing, functional and performance, could save more than \$20 billion (1) every year. The focus in this paper is specifically on performance defects.

Does it make good project sense, though, for development teams to shoulder the responsibility for finding all of a system's performance defects? Do developers have that 'second sense' that somehow leads them to the heart of the bottleneck? Do they have the time and bandwidth required to learn and operate today's complex load generators? Do they have the time or expertise to instrument a system with just enough monitoring to find the problems, but not so much monitoring that the monitoring overhead drags down actual performance?

This paper argues that developers have neither the expertise nor the time to meet these challenges themselves. They are simply not properly equipped and this has been a road block to better, earlier performance testing. Not all performance defects, though, require decades of expertise to find. Not all performance defects are enigmas. In fact, the author has seen that many problems are easy to find and fix. Have you ever seen the same type of performance problem show up in project after project? A missing database index is a good example.

Locating, managing and fixing these less difficult defects is tedious work. Couldn't this more tedious work be offloaded from SPE (System Performance Engineering) teams to software developers in an effort to fix some of the performance defects earlier in the cycle? Couldn't a line be drawn in the sand to divide the responsibilities? Developers could be responsible for locating a well-defined set of commonly found performance defects while SPE teams could be responsible for finding all the uncommon variety.

Once this work is successfully delegated and offloaded to developers, SPE teams will have the bandwidth to tackle new and more complex performance challenges.

## Introduction

The body of this paper details a testing regimen that is carefully distilled to include only those performance defects that meet two simple criteria:

- 1) Commonly occurring performance problems. SPE engineers encounter these types of performance defects in tuning project after tuning project.
- 2) Easy to find performance problems. SPE engineers can succinctly prescribe some easy-to-implement

---

1 [http://www.computerworld.com/s/article/72245/Study\\_Buggy\\_software\\_costs\\_users\\_vendors\\_nearly\\_60B\\_annually](http://www.computerworld.com/s/article/72245/Study_Buggy_software_costs_users_vendors_nearly_60B_annually)

techniques for locating these defects.

Since a one-size-fits-all regimen for "testing early" is unlikely to exist, this definition has been intentionally crafted to be vague. For instance, device driver performance is different from the performance of both decision support systems (DSS) and online transaction processing (OLTP) systems. However, developing a core testing regimen for each type of system (device drivers, DSS, OLTP, etc.) could further the endeavor of testing earlier to reduce costs.

Although the criteria for this regimen are somewhat vague, there are some common threads. For instance, the testing measures all rely on simple, open-source or custom written tools to keep licensing costs from becoming a road block. Also, most of the techniques prescribed here do not even require significant load. Instead, they rely on simple, garden variety unit tests and reports that highlight a few specific, ubiquitous inefficiencies.

The particular testing regimen presented in the paper is tailored for OLTP Enterprise Java business applications.

## **Testing Regimen Overview**

Early in development, system efficiency is a big unknown. It is a dark cloud hanging over the project. Will this system be efficient enough? Early on, there is not enough data to say. This test regimen begins to fix that. It delivers hard efficiency data where it is much needed: to the developers early in the development cycle.

The testing regimen addresses a number of performance concerns that resurface, in the author's experience, year after year in performance tuning projects. The reader is encouraged to expand and enhance this list of concerns that are both "commonly occurring" and "easy to find."

There are two parts to the testing regimen. The first part is a set of reports that details specific efficiency metrics. The second part is a set of unit tests that provide much-needed visibility into key parts of your system that are prone to performance defects.

Here is an outline of the performance concerns. Pay close attention to the reports for duplicate requests. Fixing these problems can arguably have the biggest impact of all these techniques.

### **EFFICIENCY METRIC REPORTING**

#1: Duplicate Request Reports for SQL and Other Resources

#2: SQL Statement Efficiency Reports

#3: Report for Duplicate HTTP Requests

### **AUTOMATED UNIT TESTING**

#4: No-load Response Time Regression

#5: Longevity Testing

#6: Functional Tests Required for Efficiency

## **Efficiency Metric #1: Duplicate Request Reports for SQL and Other Resources**

"Avoid duplicate, unnecessary processing." This simple directive is pretty straight forward, but it is also so vague that it is tough to follow. However, this can be remedied if you create automated, daily reports that tally the counts of requests submitted to various back-end systems. One report should be created for each business process.

SQL requests to RDBMS are the primary focus, but any resource that can be deployed over a network is a concern. Here are some examples of various types of requests that you should report on:

- RDBMS (Relational Database Management System)
- SOA (Service Oriented Architecture)
- CICS (Customer Information Control System)
- JMS (Java Messaging Service)
- LDAP (Lightweight Directory Access Protocol)

- Ajax or web

These reports can often be generated by high-dollar Application Performance Monitoring (APM) tools like CompuWare's DynaTrace or CA's Introscope (see Resources section for details). SPE teams are accustomed to budgeting for high-dollar tools like this. Development organizations are not. So, the following section details one possible approach to creating these reports without an expensive tool.

## CENTRALIZED LOGGING

These reports require that a single log entry be written for each "back-end" request. Here are some recommendations for making this take easier.

- To avoid having to write and maintain custom logging code, try to find an existing logging facility for the backend system you are tracing.
- If custom logging is required, add the logging code in easy-to-maintain centralized locations, instead of at the many points in the application code where the requests are made.

For instance, JDBC activity is easy to monitor. Just use a JDBC driver "wrapper" like P6Spy. It implements the JDBC interface and delegates all functionality to your real JDBC driver. It is intended specifically for logging SQL statement invocations with no code changes at all. The Resources section contains detail on P6Spy and two other options for SQL monitoring.

## CACHING CAUTION

Performance savvy engineers cache the results of certain SQL SELECT statements whose underlying data rarely changes. Why incur the overhead of a roundtrip to the database if the response data will be identical for every request? The "Resources" section mentions a few great tools like EHCACHE and JCS that can be used to implement caching. A detailed look at caching is beyond the scope of this paper.

Unfortunately, though, caching makes a "Duplicate Request" report a little more complicated. The cautionary tale is this: SQL statements and other activity for the report should only be captured *after* these caches are filled (aka "warmed up"), to avoid SQL statement counts that do not accurately reflect normal system behavior.

Warming up the caches is helpful for almost all performance unit testing, not just the Duplicate Request reports. So, be sure to thoroughly warm up the system prior to unit testing, especially automated testing. Here is one approach to try: run the entire automated testing process twice, discarding the results from the first iteration.

The magic of caching happens under the covers of the system, where few ever witness cache hit, cache miss and cache expiration events. Are all these cache events happening when you need them to? The section in this paper titled "Functional Tests Code Required for Efficiency" details functional testing that will validate that these events are happening at the right times to optimize performance.

## BIND VARIABLES

This section shows how to use "bind variables" in the logged SQL statements so that unique SQL statements are correctly identified. The goal of the Duplicate Request Report for RDBMS is to capture all the SQL statements executed during a single business process into a text file. Note that question marks (?) need to be used instead of the actual values in the SQL statement. Without this, the tools used below for sorting, counting and grouping will see multiple variants of the same SQL statement as completely different statements. For example, consider these two SELECT statements:

```
SELECT A FROM B WHERE C=X
SELECT A FROM B WHERE C=Y
```

If X and Y, above, are instead represented as a single character (normally a question mark) like this:

```
SELECT A FROM B WHERE C=?
SELECT A FROM B WHERE C=?
```

then both of these statements can be correctly identified as a single statement as opposed to two different ones. Each question mark represents a "bind variable."

## THE REPORT

Here is an example of some SQL statements captured in a text file from a single business process:

```
SELECT * FROM CUSTOMER WHERE CID="?"
SELECT * FROM CUSTOMER WHERE CID="?"
SELECT PRD_NAME, AVAIL_DATE FROM PRODUCT WHERE PRD_ID = '?'
SELECT * FROM PARAMETER
SELECT * FROM CUSTOMER WHERE CID="?"
SELECT * FROM PARAMETER
SELECT * FROM USER WHERE USER_ID="?"
```

To sort and count this data in a file named mySql.txt, you can use this very simple UNIX command. See the Resources section for similar approaches for Windows environments.

```
sort mySql.txt | uniq -c
```

The above command yields the following text which shows how many times each SQL statement was invoked:

```
3      SELECT * FROM CUSTOMER WHERE CID="?"
2      SELECT * FROM PARAMETER
1      SELECT PRD_NAME, AVAIL_DATE FROM PRODUCT WHERE PRD_ID = '?'
1      SELECT * FROM USER WHERE USER_ID="?"
```

Voila! The generation of the report should be an automated part of the testing regimen.

Once the report is readily available to developers, some analysis has to take place before performance problems can be fixed. Here is an example:

*This little report shows us that the application makes three repetitive-looking round trips to the CUSTOMER table. If all three of these queries look for the exact same CID, then it is likely that two of the three queries are unnecessary and the application can be re-factored to work without these extra queries. If instead, all three queries retrieve unique customer records, then perhaps the SELECT statement could be re-factored to retrieve data for all three customers in a single round trip, like this: SELECT \* FROM CUSTOMER WHERE CID IN ('A', 'B', 'C').*

Without too much difficulty, this little report could be enhanced to include the average run time of each query, in addition to the count.

## CUSTOM ACTIVITY LOGGING

As mentioned above, it is helpful to rely on some existing facility to log SQL or other activity. A backup plan is required, though, when this logging isn't available or expedient. After attempts have been exhausted to find a tool or utility to log activity (like P6Spy, above), consider adding custom logging code in a centralized, architectural component.

This section simply shows how logging with a single sentinel string for the custom logging of all resource types makes the reporting chore a little easier.

Say that you need logging for a request to a 3<sup>rd</sup> party Web Services system. Choose a "sentinel" character string to be part of every log message, regardless of the logging type (CICS, JMS, LDAP, etc). A simple text search through the application logs for this sentinel string makes it easy to create a report for all activity, regardless of the logging type. The following class provides a simple example:

```
import org.apache.log4j.Logger;

/**
```

```

* Architectural component used to submit request to 3rd party Web Services System
* Debug logging is used with "~~##" as a sentinel string for easy text searching.
*/
public class WebSvcRequestUtil {
    private static Logger _log = Logger.getLogger("WebSvcRequestUtil");
    public static void remoteRequet(String strRequestName, String strRequestXML) {
        _log.debug("~~## WebService request name:" + strRequestName);
        submitRequestRequest(strRequestXML); //pseudo code to submit the request
    }
}

```

The `log.debug()` statement uses a special character string (“~~##”) that will be later used to locate (using `grep` or `findstr`) this custom log statement in the application log file.

```
_log.debug("~~## WebService request" + strRequestName); //simply copied from above code snippet
```

If custom logging is required for additional resources, use the same ~~## sentinel string as shown below. Note that each of the three components detailed writes to a single log file on the application server.

```

//Custom logging in a JMS architectural component
    jmsLogger.debug("~~## JMS request:" + strRequestName);

//Custom logging in an LDAP architectural component:
    ldapLogger.debug("~~## LDAP request:" + strRequestName);

//Custom logging in a CICS architectural component:
    cicsLogger.debug("~~## CICS request:" + strRequestName);

```

After a full business process is executed, a simply search through the log file for the sentinel string (~~##). Then sort, group and count the results as shown above, using the Unix “`sort`” and “`uniq`” commands.

## Efficiency Metric #2: SQL Statement Efficiency Reports

### SQL STATEMENT ANALYSIS PART 1

The report detailed in the above section counted the number of invocations of each unique SQL statement. The following report, instead, dissects the text of each SQL statement, highlighting various practices that are prone to poor performance:

- Large number of tables in the FROM clause.
- Large number of columns in the SELECT LIST.
- Large number of criteria in the WHERE and other clauses.

When reporting on counts of SQL invocations, it was helpful to look at only the SQL for a single business process. It allowed us to ask questions like this: “Does this single business process have a need to go to the same table two different times?” The story is different for this report. The three anti-patterns detailed above can cause problems regardless of which business process they belong to.

So, this report should not be grouped by business process. Instead, it should list just the worst offenders of each of the three metrics for all SQL for the entire application. To limit focus to only the worst issues, perhaps the results should be limited to the 10% with the highest counts in each of the three detailed metrics.

Code will need to be written to do the light parsing work to produce the above three counts.

### SQL STATEMENT ANALYSIS PART 2: FLAG RDBMS DATA TYPES PRONE TO PERFORMANCE DEFECTS

Automated unit tests should report on all tables that use data types that are prone to performance problems. BLOBs immediately come to mind, along with similar types like CLOBs. These stand for “Binary Large Object” and “Character Large Object.” Simple queries to the database’s “catalog” tables quickly display the names of all tables that use these data types. See the Resources section for details on these catalog tables.

Why are BLOBs so prone to performance problems? Databases often require multiple round-trips to access these special data types. These multiple round-trips can significantly hurt performance, especially when multiple BLOB rows are returned in a single result test.

It often happens that only a small percentage (say 10%) of the BLOB or CLOB data in a table is actually long enough to require a BLOB or CLOB type. The rest of the data fits into a more traditional and more significantly more efficient data type like VARCHAR.

Why should your system incur the BLOB overhead for a large percentage of data that does not require a BLOB? Consider the following optimization, which only incurs BLOB performance overhead for the small percentage of data that actually requires it.

Alongside the BLOB field, in the same table, define a new VARCHAR column named "short\_blob." Define this new column with the maximum width allowed.

The variable length short\_blob column will be used for all data that is short enough to not require a BLOB or CLOB. Store all other data in the BLOB. Then, re-factor all application code with this behavior:

- For a single row in the table, the code will use either the short\_blob column or the BLOB column but never both.
- Only insert data into the BLOB/CLOB column if the data does not fit into the short\_blob column.
- When SELECTing from this table, the code should first check the short\_blob for data. If the data is null, then the code should take the performance hit of retrieving the data from the CLOB/BLOB.
- UPDATES to the data will require a check to see if a data length changes requires a switch from the short\_blob column to the BLOB column.

### SQL STATEMENT ANALYSIS Part 3: Minimize SELECTs to "static" data

With the proper caching in place, SELECTs to certain RDBMS tables can be nearly eliminated. A simple report, automatically generated every day, can easily highlight any tuning opportunities by showing SELECTs to this certain set of tables.

- 1) Start by identifying tables that only contain data that infrequently changes. More specifically, get lists of all table names that only contain data that is only INSERTed, UPDATED or DELETED a dozen times a day (or less). These tables will be referred to as "static" tables. Below are some examples of "static" data. Remember that caches can be configured to expire so that the data is refreshed at a particular interval.
  - a. Data that defines products, like cell phone billing plans or savings accounts.
  - b. Organizational hierarchy information and lists of physical locations.
  - c. Names of cities/states/regions/provinces.
- 2) Create an automated report that displays all SELECT statements to the list of "static" tables, created in the above step.
- 3) This report will create tons of "false positives" if the recommendations in the section titled "CACHING CAUTION" are not followed.
- 4) Work to improve caching code until there are zero entries on this report.
- 5) Here is one helpful enhancement to this report: show the counts of SELECTs to each static table.

### FETCH SIZE ANALYSIS

The "Fetchsize Analysis" detailed in this section is, unfortunately not something that can be easily automated. However, for the small amount of work required, this analysis can bring great efficiencies.

How many trips to a JDBC database does it take to retrieve a single result set? How much memory is pre-allocated by the JDBC driver for data retrieved by a result set? The JDBC "fetchsize" parameter controls these factors. However, judging by how seldom this parameter used in application code, this parameter is a well kept secret.

The fetchsize basically controls how "chatty" the JDBC client is with the database server. Specifically, it states how many DB rows are returned in a single round trip to the db. For example, if your fetch size is 1 and your code iterates through 11 rows, then your DB will make 11 round trips to the database (often over the network) to retrieve all the data.

If the fetch size is 20 and the result set has 40 rows, then 2 round trips are made.

Here are some recommendations for tuning the "fetchsize": Locate all SQL queries that identify a single row of

data from the database – in other words, all things that use primary key lookup. Then, set the `java.sql.Statement's` (or `PreparedStatement's`) `fetchsize` to 1 for each of these queries that returns a single row. An example is included in the Resources section.

When `fetchsize` is smaller, less memory is allocated for the resultset and the java garbage collector is required to do less work, lowering system overhead.

For queries that return larger result sets (like a screen that displays an account history), be sure to set a larger fetch size. This will reduce the number of round trips to the database and improve performance.

### Efficiency Metric #3: Report for Duplicate HTTP Requests

The section “#1: Duplicate Request Reports for SQL and Other Resources” detailed reports that show, among other things, duplicate invocations of SQL statements. This same kind of report can also be created to identify duplicate HTTP requests.

However, producing this report for HTTP requests will be a bit more difficult than it was for SQL statements (detailed above). Earlier in this paper, we established that techniques in the testing regimen must be both “commonly occurring” and “easy to find.”

In fact, getting the HTTP data will be difficult enough that it violates the “easy to find” guideline. Consider these difficulties:

- It was easy to dump SQL statements into raw text files like the SQL statements. The P6Spy driver did that for us. More work, however, is required to convert HTTP data into some format (like a text file, for example) that is more conducive to sorting, grouping and counting. HTTP and TCP tracing tools like Fiddler2, WireShark and tcpdump capture data in one format and must be somehow converted to text. See the **Resources** section for more detail.
- When tracing SQL statements, the request-specific data values (called “bind variables”) were already removed by the SQL tracing tool (P6Spy). More work is required, though, to remove these request-specific values from HTTP requests.

If these two processes were in place, it would not take much effort to automate an HTTP Duplicate Request report. But without these conveniences, the work must be done manually.

The free tool Fiddler2 (see the Resources section) is just one of many tools that can be used to easily trace traffic from a browser. Fiddler2 can also be used, even when the HTTP traffic is between two servers, instead of a browser and a server. However, this requires that you reconfigure your system with Fiddler2 as a proxy server.

The following technique, however, provides a large convenience: it allows you to trace any HTTP traffic without changing any system configuration. See the Resources section for details on the tcpdump utility. Two other tools, WireShark and the “Web Performance Analyzer”, can both be used to analyze the tcpdump data.

Because there is an understanding of what it will take to automate this report, the technique is still included here:

- 1) Say that you have a JSP application the sends messages back and forth to a SOAP server. Both applications are deployed on UNIX machines. A small amount of work can provide a trace of the entire conversation between the web server and the SOA server.
- 2) Create a tcpdump of the traffic back and forth between the web server and the soap server. All steps should be done on either the web server or the SOA server.
  - a. Create a text file on the Web server called `filter.txt` with just this text. Say that the web server is named “webserver.com” and the soap server is named “soapserver.com”.

```
(src host webserver.com and dst host soapserver.com) or (src host soapserver.com and dst host webserver.com)
```
  - b. Run the following command:

```
tcpdump -F filter.txt -i eth0 -s 65535 -w myFile.pcap
```
  - c. Execute the business process while no other traffic is executing on the servers.
  - d. Stop the trace by issuing the `Ctrl+C` command.

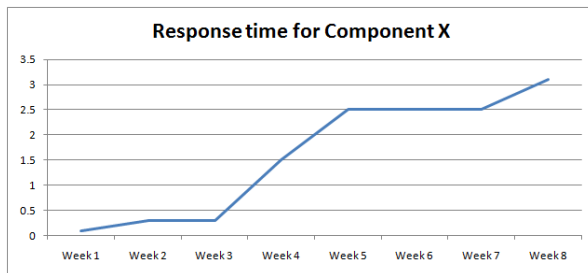
- e. Download the myFile.pcap file to your Windows desktop.
  - f. Use the “Web Performance Analyzer” or WireShark to view all the traffic between the two systems. Look for unnecessary messages that can be eliminated.
- 3) Use the above tcpdump technique to look for message consolidation opportunities. Watch for multiple requests submitted in rapid-fire succession, especially those that are sequentially and synchronously executed. Consider refactoring the code so that the multiple requests all execute on the server instead of the client.

#### Automated Unit Test #4: No-load Response Time Regression

A Performance Service Level Agreement (SLA) for response time sets expectations for how fast the application should process a request in production to meet customer requirements. It would be nice if automated performance unit tests could compare test response time to the production SLA. However, there are many factors why response time in a test environment varies from that in production. The number of network hops between various servers is often different between test and production. The size of result sets for various inquires varies greatly between test data sets and those in production.

For these and other reasons (like CPU, RAM, disk and firewall differences), comparing production response times to test is a questionable practice, at best. However, an automated performance testing process can still compare today’s response time to that of yesterday. That process can still brandish big red flags when response time recorded this week is three times as bad as the previous week.

This paper recommends tracking response time in unit tests from the very first day the system utters “Hello world.” That way, response time leaps in data collected over weeks and months can easily be attributed to various project milestones, source code commits and changes in test data.



**Figure 1: Response time for one particular component, collected from the beginning of the project. This graph shows that changes introduced in weeks 3 and 7 caused response time to degrade.**

#### Automated Unit Test #5: Longevity Testing

One way to detect a memory leak is to look for an upward trend in memory consumption over an extended timeframe, perhaps 24 hours. This kind of test is common place for a performance testing effort, right before an application goes into production.

Running the longevity test immediately prior to production is a good idea; it really is. But what is keeping us from also running longevity tests earlier in the development cycle? When a few pieces of the architecture first find a measure of stability, how difficult would it be to test these components repeatedly for 24 or 48 hours against a single instance (2) of a virtual machine?

In fact, surviving a longevity test should be a requirement for delivering an architectural component.

If the entire concept of an early longevity test is overwhelming, then just start simple. Start with activity of a single user executing requests. Then, enhance the test over time to apply more load. It is important to maintain a steady load for the duration of this test, due to concerns detailed in the next section.

## LONGEVITY TESTING EXIT CRITERIA

Obviously, the longevity test fails if server crashes. This section details other behavior to validate.

Validate steady state behavior. The load in this test, even if it is a single user or test driver, should remain steady for the duration of the test. Since the load is steady, the following metrics should also remain steady over the course of the test:

- JVM heap consumption
- Operating system RAM consumption
- CPU consumption
- Response time

Today, very little longevity testing happens early in the development cycle. An automated approach to validating the above metrics, which is outside the scope of this paper, would make it easier for development to adopt and embrace longevity testing.

## Automated Unit Test #6: Functional Tests Required for Efficiency

There are certain parts of a system that are used frequently in production but are only seldom actually seen by anyone. These obscure parts of the system often do not get enough testing.

Unfortunately, some aspects of system performance heavily rely on exactly these types of features, ones that operate quietly under the covers and are rarely tested.

## FUNCTIONAL TESTS TO VALIDATE DATABASE INDEXES

Developers could really use an automated JUnit-like framework that validates that database indexes are in place for all SQL statements. Here is an outline of the testing:

- 4) Run a business process (in isolation).
- 5) Capture all SQL statements
- 6) For each SQL statement, get the database's opinion of whether the statement is indexed properly.
  - a. For DB2, use the db2advise tool. This tool tells you whether your SQL statement is indexed properly. But the tool does not stop there. If it is not indexed properly, it will even generate a "CREATE INDEX" statement that will clean things up.
  - b. Oracle provides with a performance report called "awrsqrpt.sql" that detailed indexing and other performance metrics for single SQL statement.

## FUNCTIONAL TESTS TO VALIDATE CACHING HITS, MISSES AND EXPIRIATION

Very few automated unit tests exist to validate that caching functionality works correctly. More specifically, tests are required to verify that cache hits, misses and expiration events happen that the correct times. The section

---

2 Of course, re-starting the JVM repeatedly would be like destroying evidence at a crime scene for every test. Instead, the test must determine whether a single running instance of a virtual machine can withstand a load for an extended timeframe.

titled "SQL STATEMENT ANALYSIS Part 3" provides more detail on caching.

A full description of caching is beyond the scope of this paper. But the Resources section of this paper references a few great caching tutorials.

Caching functionality can be added to inquiries on any type of resource, not just RDBMS. Here is an overview of the testing for testing SQL caching.

- 1) Validate cache miss: Immediately after system startup, the cache should be empty. Execute your cache-enabled application component that performs the SELECT/inquiry. Validate that the SELECT statement correctly retrieves the data from the database.
- 2) Execute the cache-enabled component a second time. Validate that the results are retrieved from the in-memory cache instead of from the database.
- 3) Write test code to force the memory cache to be emptied/expired. Then repeat step 1.

## Discussion

### ARCHITECTURE FIRST

Having response time and throughput SLAs early in development is a rare. If performance SLAs are available, then by all means, use them. But do not let the lack of SLAs keep you from performance testing. Most of the elements of this testing regimen show the progress that can be made without load and without SLAs.

Throughput goals generally govern the deliverables of the completed application. By definition, though, the completed software is not available early in the development cycle. Presumably, though, significant parts of the architecture should be available early on.

So, the architecture should be the first focus of early performance testing.

Furthermore, early performance gains made in the architecture will benefit all the business processes that use the architecture. Test the architecture first. How should the architecture be tested? As detailed above, an architectural component should not be considered complete until it passes a longevity test. See the section titled "**#5: Longevity Testing**" for details.

Where is the dividing line between architecture and application functionality? Early performance tests could rely on test programs used to development architecture. Or, a few core business processes could be used to drive the architecture. Also insure that each major architectural component gets tested.

### BE WARY OF EXCESSIVE LOAD

It is admirable to test components under load early in the development cycle. This is tough to deny. However, it is a waste of time to fix problems caused by a particular load that is higher than the system is expected to handle. Performance unit tests, especially ones testing small, granular components, often make this mistake.

### WHEN

When should this testing be done? Whenever you can! Here are some convenient spots in the project plan to do developer-driven performance tests:

- 1) When code is first laid down that connects two tiers.
  - a. Middle tier and the database
  - b. Middle tier and a 3<sup>rd</sup> party SOA provider
  - c. GUI and the middle tier
- 2) First complete business process. For many applications, when the first business process is completed, many of the architectural components are getting used at least one time. This is, therefore, a great time to implement some performance testing.
- 3) When additional business processes get completed.

## PACKAGING

Just like functional unit tests, performance unit tests should be deployed right alongside production code. This helps to validate whether performance problems at a new installation can be attributed to the environment conditions, such as network firewall latency.

### A SMALL, DEDICATED PERFORMANCE ENVIRONMENT

Time and effort will be well-spent obtaining a small, dedicated environment that is controlled and isolated. Only unit tests should be exercising this environment. Consider deploying the code and a database together on a single, inexpensive system. Having enough RAM is a higher priority than having a fast, high-powered CPU.

This test regimen will work well with a lower-powered environment because it de-emphasizes the need for heavy load. A small, dedicated environment also enables the performance tests and reporting to focus on a single, unmoving version of the code, isolated from developer's hourly code and data changes. Also, this regimen does not require that load be generated from a separate machine, like with traditional SPE performance tuning efforts.

Periodically re-deploying new code to this performance environment takes time, though. Organizations with a mature, automated redeploy process will have an obvious advantage here. In addition to the redeploy, a performance environment like this will greatly benefit from an automated process to reload the database(s) with a known set of data. That set of data should be kept up-to-date with the code release being tested.

The entire redeploy process, for code and data, needs to be automate and repeatable to minimize the burden on the development team.

### Summary

To summarize, developers face a number of road blocks when trying to locate performance defects early in the development cycle. For instance, this paper argues that developers do not have the expertise find their own performance defects on their own. They do not know which performance metrics are the most critical. They traditionally do not have licenses for the tools (APM tools) that provide these metrics and they do not have the time to master these complex tools.

However, the testing regimen detailed herein partly addresses these issues. It essentially pre-packages some tuning expertise that can be thrown over the wall. It shows developers which metrics are critical, along with instructions for getting those metrics.

Keep in mind, though, that some report coding and unit test coding is required. Also, developers will need to acquire small, dedicated performance environments to implement this testing regimen. Furthermore, this testing regimen is not designed to find all performance defects – just those that are both commonly found and easy to locate. Defects found exclusively under a heavy load, for instance, are not addressed in this paper. Locating these and other complicated defects will continue to be the realm of SPE teams.

The testing regimen detailed herein is specifically tailored for OLTP Java Enterprise business applications. There are many other types of applications that will require their own tailored performance testing regimens: Business Intelligence, console gaming, embedded systems, etc...

The work required to implement the testing regimen is all detailed here. Funding this work is small price to pay for the significant cost reductions that can be realized by finding performance defects earlier in the software development cycle.

### Conclusion

What is next? What else can be done to reduce costs and further mature the software we deliver? As mentioned, this one performance testing regimen is for just one type of software (Java OLTP systems). Other similar performance testing regimens need to be created and implemented for other types of software: device drivers, manufacturing command and control systems, operating systems, mobile devices, etc.

Once SPE teams distill their experience into concrete best practices, they must not keep it to themselves. That experience must be somehow passed on to development teams. There are a number of ways to do this:

- Introduce developers to testing regimens, like this one.
- Implement training programs where SPE teams mentor developers.
- Implement new “Entrance Criteria” that stipulates that certain performance testing (like that found in this testing regimen) must be completed before SPE teams begin tuning an application.
- Get an “Executive Sponsor” that will help shepherd the cause of testing earlier for performance defects.
- Get developer-driven performance testing on the project schedule, so that milestones are being tracked, and developers are getting nagged: “Is it done yet? Is it done yet?”
- Change jobs. SPE technicians could leave their positions in testing organizations to bring performance testing experience to development organizations. The author of this paper, in fact, made this exact job transition while writing this paper. Perhaps this job change will provide enough fodder for a case study at CMG 2012.

## Glossary

**APM Tools** – Application Performance Monitoring tools

**DSS** – Decision Support Systems

**IN ISOLATION** – A term used to describe system activity that is executed without any other activity. Running a business process in isolation is helpful in discovering all the events triggered by a single business process.

**OLTP** – Online Transaction Processing Systems

**RDBMS** – Relational Database Management System. An RDBMS is a database that processes SQL requests.

**STATIC DATA** – A term used to describe data in a database that only infrequently changes, perhaps a few times a day.

**SPE** – System Performance Engineering, performance tuning specialists.

**SOA** – Services Oriented Architecture

## Resources

JDBC FETCHSIZE -- Use this fetchsize [tutorial](#) to minimize:

- the number of round-trips to the database
- the amount of RAM allocated by the JDBC driver for result set data.

## FREE TCP/HTTP TRACING AND ANALYSIS TOOLS

- [WireShark](#) – WireShark shows extraordinary trace detail, but does not show a timeline of activity.
- tcpdump – a TCP trace facility distributed with most UNIX installations. This can be [downloaded](#) for Windows (called WinDump). This provides no facility for analyzing the trace.
- [Web Performance Analyzer](#) – helps to visualize a timeline of HTTP requests. This tool reads trace files created by by WireShark and tcpdump (the pcap format).

## APPLICATION PERFORMANCE MONITORING (APM) TOOLS

These tools cost tens of thousands of dollars to license, but can easily generate some of the reports detailed in this paper. Here are two of the many available:

- [CompuWare DynaTrace](#)
- [CA Introscope](#)

LOG4J:

A [popular java API](#) used for logging data used in the Duplicate Request Report detailed in this paper.

## SORTING, GROUPING AND COUNTING TEXT DATA

- UNIX: “cat myFile.txt | uniq -c”
- Windows: Use ODBC to SELECT/COUNT/GROUP text stored in a file. [Link 1](#). [Link 2](#) [Link 3](#).

## OPEN-SOURCE JAVA CACHING

- [Documentation](#) for EHCache; [Read this tutorial on caching](#) (requires registration).
- [This monitoring](#) can be used to monitor cache hit/miss/expiration events in production.

- High-level caching discussions: [link1](#) [link2](#); An introduction to JCS cache: [link](#).
- A nice comparison of various open-source tools that can be used for caching: [link](#).

#### DATABASE CATALOG TABLES

Write JDBC code to query Oracle [system tables](#) to discover whether your schema uses data types that have traditionally had many performance problems.

SQL STATEMENT LOGGING OPTIONS: This logging is required for the Duplicate Request reports.

- Use the JDBC driver wrapper [P6Spy](#).
- Query the Oracle [V\\$SQLTEXT](#) table for a lot of all SQL statements executed.
- Use a relational persistence layer, like [Hibernate](#), which provides decent SQL statement logging.

GREP and FINDSTR -- Use these are UNIX/windows tools to find a particular text string in a text file.