

# High Performance Java Development : It's the Database, Stupid

By Timothy Fox  
Enkitech, Inc



Most corporations today are standardizing on “n-tier” application architectures with browser-based user interfaces. This is due, in large part, to the availability of network bandwidth along with the need for “access from anywhere.” Sun’s Java 2 Enterprise Edition (J2EE) is one of the two main architectures of choice (the other being .Net) across industries for supporting such business requirements. In the largest, most critical systems, the Oracle database is often chosen for all the obvious reasons – high performance, scalability, tunability, etc.

This paper will focus on methodologies for building high performance, business critical applications using the J2EE / Oracle stack. It is no secret that Oracle’s WebLogic server is recognized as the standard for middle tier application servers, but the application server brand, for the purpose of this document, is interchangeable as the details of configuration of such products will not be covered. What will be covered are techniques which enable the developer to see what their database code is doing in the event the performance becomes an issue.

When building a business application (we’re excluding games, utilities, etc.), the user is generally making a request using an HTML based interface and is provided a response which is populated with information from a database. Take a Call Center application, for example. The diagram below shows a typical conversation:

|

## Customer Service Workflow Steps

1. The customer contacts the call center to pay a bill

Steps 2-4 are executed repeatedly throughout the conversation with the customer

2. The Call Center Agent performs a search for the customer's account information
3. The search is performed on the database server and the pertinent data is returned to the application server
4. The data is formatted such that it can be rendered on the browser and sent back to the requestor (Customer Service Agent's browser)
5. The Call Center Agent reports the success of the transaction to the customer and the process ends

Consider a Call Center which employs around 500 agents (typical for larger centers). If the application generates multiple database requests per user transaction, this could translate into hundreds or thousands of simultaneous requests, all of which result in database activity.

As stated in the title of this paper, it's the database that is doing all the heavy lifting in this type of application. The Web server farm is merely brokering JSP requests and possibly caching some static content. The Application Server farm is responsible for making connections to the database, receiving data, and formatting the output for the browser.

The Database, however, is potentially performing a tremendous number of transactions against tables containing millions of rows. It is this scenario that should cause all development teams to hone their SQL and PL\*SQL skills as even a single run-away query can drag system performance below acceptable levels. It is common, and at the same time unfortunate, to see development teams which are fluent in Object Oriented concepts, but are unprepared to write and tune SQL code which performs well under load. Although the concepts discussed here are universal, the examples will be specific to an Oracle database.

A vital component of any J2EE-based, enterprise application is the connection pool. Not all connection pooling systems work the same way, so it is very important to understand the characteristics of the one(s) you are using. A mis-configured connection pooling mechanism can in some cases do more harm than good.

## High Performance J2EE Applications – Where to Focus?

### Brand New Applications – Focus on Database Design

If you have the luxury of starting a project from scratch, many performance problems can be averted by starting with a sound database structure. It is impossible to say that one concept for database design works better than another without assessing how the application will exercise the database. Knowing the type and size of the user population along with reporting and batch requirements will allow the data architect to build a schema which will support the business requirements while also taking future scalability into account.

In terms of Java development, focus on doing what Java does well, formatting output for the user. Allow the database to do what it does best – manipulate data. The database server will be much more powerful (more CPU's and more memory) than an Application Server so try to avoid doing massive data manipulation in the middle tier.

### Existing Applications – Find and Fix problems, Write Good Code

If you're not the first developer on the scene (which is probably more typical), then your job will likely entail fixing problems while pouring more code on top of what already exists. The key to this role is knowing how to use the tools at your disposal to succinctly diagnose issues and formulate resolutions. We will review the following tools / concepts:

#### Java Virtual Machines (JVM)

- Java Virtual Machine Heap Management
- Java Virtual Machine Monitoring
- Connection Pooling

#### Oracle Database

- Using StatsPack / AWR Reports to pinpoint problems
- Looking at "actual"
- SQL Tuning

This paper will not teach you to be a tuning expert (that takes years of experience), but will introduce you to the tools you need to identify your issues.

## Java Virtual Machine (JVM) Heap Management

The JVM is actually a virtual computer running on your hardware. Where your hardware has physical memory, the JVM has the “heap” which is the memory in which the JVM runs. The heap has to support both the JVM itself along with any code you write. The amount of memory you can allocate to the heap depends on your operating system:

Operating System	Recommended Heap Size	Maximum Heap Size
32 bit	1.5MB	2.0GB
64 bit	2-3G	4.0GB

As you can see, even the maximum heap size is relatively small compared to the physical server’s memory capacity. This means that you must fit your application into a fairly small memory footprint or you may experience JVM performance issues due to excessive Garbage Collection (more on this in the next section).

Here are techniques to help manage heap usage in a J2EE application:

1. Close everything that you open (especially database connections)

Here are two examples of opening and closing a database connection – one good and one bad:

### Good One

```
ResultSet rs;
PreparedStatement ps;
makeConnection();
ps=Conn.prepareStatement(q);
rs=ps.executeQuery();
while (rs.next()){
    do something;
}
rs.close();
ps.close();
Conn.close();
```

### Bad One

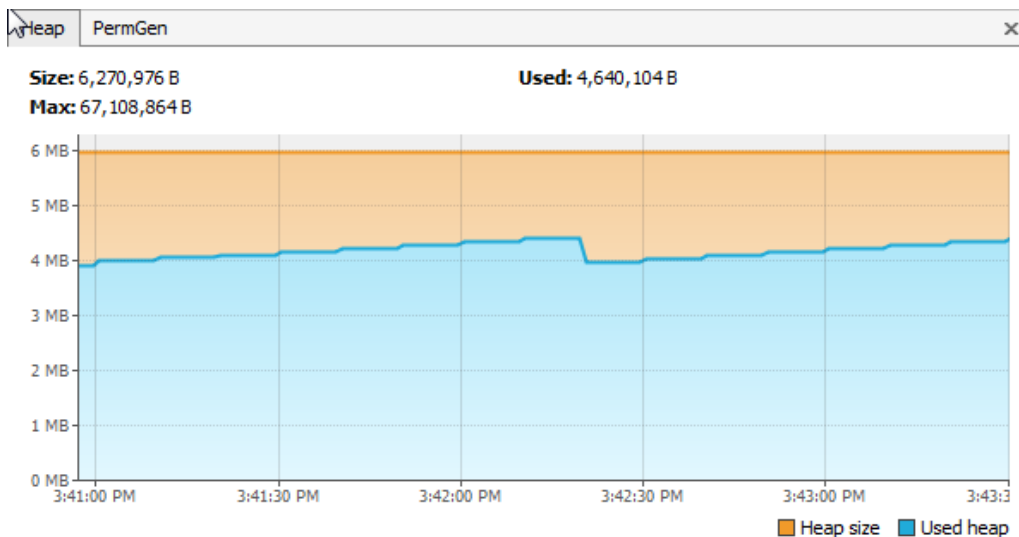
```
ResultSet rs;
PreparedStatement ps;
makeConnection();
ps=Conn.prepareStatement(q);
rs=ps.executeQuery();
while (rs.next()){
    do something;
}
Conn.close();
```

The difference between these two code blocks is that the “Good One” closes all of the objects and does not assume, as in the “Bad One,” that the close connection removes the result set and prepared statement from memory. Although the orphaned objects may eventually be cleaned up by a Garbage Collection, it is better to close or remove anything that you open or create.

## 2. Don't cache too much stuff in the JVM's heap

Keeping in mind that the heap is used not only by your code but also by the JVM itself, you are not even allowed to use the whole amount of memory you asked for. For this reason, caching large amounts of data in memory is not recommended.

The screen shot below shows the profile of a Sun 1.5.0\_15 JVM hosting a Tomcat 5.5.14 Application Server. This instance of Tomcat was not accessed during the time shown, but you can see the memory profile climbing and just before the 03:42:00pm mark, a Garbage Collection occurred.



What this graph shows is that the JVM is constantly using its heap and performing Garbage Collections. If an application uses this memory space cache objects, the JVM may be forced to clean up after

### 3. Don't do work that the database can do

If you have Java code that calls the database, receives the results, and then calls the database again using the something from the previous result set, try to consolidate this activity. Making multiple round trips to the database can become an expensive activity when processing hundreds of requests per second.

Remember that Oracle is likely running on a server with multiple CPU's and tens or hundreds of gigabytes of memory, all of which it can use. Compare this to the 1.5 to 4 gigabytes of memory allocated to a JVM and it becomes obvious that the database is a tremendous resource for your application.