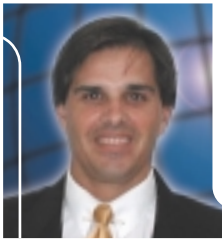


Best practices

WebSphere Application Server and Database Performance Tuning

BY MICHAEL S. PALLOS

**ABOUT THE AUTHOR**

Michael S. Pallos, MBA, is a senior solution architect for Candle Corp. (www.candle.com) and has 18 years of experience in the IT industry. He is a consultant to some of Candle's largest corporate customers, a featured speaker at industry conferences, and a doctoral student at Walden University. Candle, based in El Segundo, California, provides solutions to manage and optimize business-critical infrastructures, systems, service levels, and other information technology assets.

E-MAIL

michael_pallos@candle.com

Optimization of the production runtime environment boosts the performance of WebSphere Application Server applications, allowing organizations to harness the full potential of their hardware and software investments. Performance tuning of the network and database interfaces are two of the most important elements of the optimization process. This two-part series explores best practices for performance tuning as it relates to the persistence layer of WebSphere Application Server and a database management system (DBMS).

Organizations running WebSphere Application Server AE are most likely using three databases: the WebSphere Application Server configuration repository, the session persistence repository, and the application data repository. If, however, an enterprise runs the single edition version of WebSphere Application Server, it is running only two database images: one for session persistence and another for application data. The configuration repository, when running the single edition version, is contained in a single XML configuration file. Figure 1 depicts the WebSphere Application Server AE architecture with repositories. Multiple repositories increase the number of database connections and require organizations to optimize each connection independently.

The interface between WebSphere

Application Server and the DBMS(s) is critical, since this connection can either act as a bottleneck or facilitate high throughput.

Best Practices

An exploration of best practices in the following areas will provide a road map for the vital task of optimizing the storage and retrieval of persistence data:

1. Database connection pooling
2. Prepared statement cache
3. Session persistence
4. Enterprise JavaBeans
5. Java database connectivity
6. Application monitoring

We explore the first three areas in this article and will review the balance in Part 2 of the series.

Connection Pooling

Whenever an application uses a

database resource, a connection must be established, maintained, and then released when the operation is complete. These processes consume time and IT resources. The complexity of accessing data from Web applications often imposes a strain on the system. Web applications are more taxing on the system for several reasons. Specifically, Web users:

- Connect to and disconnect from the database more often than users of non-Web applications
- Normally have shorter interaction times, making the database connection the longest component in the transaction
- Participate in unpredictable usage patterns, placing more demands on the database connection

WebSphere Application Server provides connection pooling to address some of the challenges of database access. Connection pooling is the process of creating a predefined number of database connections to a single data source. This process allows multiple users to share connections without requiring each user to incur the overhead of connecting and disconnecting from the database. Connection pooling can speed up application processing significantly.

When a user makes a request to a data source, WebSphere Application Server examines the connection pool for an existing resource connection. If one exists, it is provided to the user. The system then processes the database request. Once processing is complete, the connection resource is released from the user and placed back into the connection pool.

IBM used Sun Microsystems' JDBC 2.0 Option Pack API to incorporate connection pooling. The system administrator sets the connection pool thresholds, which are easily configured using the WebSphere software administrator tools.

According to IBM's *DB2 UDB/ WebSphere Performance Tuning Guide*, best practices for connection pooling that will allow an application to achieve optimum performance include the following:

- **Use the same method to both obtain and close the connection.** This approach allows the connection resource to be released efficiently to the connection pool.

- **Minimize the number of Java Naming and Directory Interface (JNDI) lookups, an expensive process in application performance.** The JNDI expense is incurred by the out-of-process network call required for each JNDI lookup. To limit JNDI lookups, the developer should create a separate method to handle these calls. Once created, the separate method can be called from the servlets `init()` method or from an EJB's `ejbActive()` method.

- **Do not declare connections as static objects.** If a connection is declared as static, then it is possible to have the same connection used on different threads at the same time. This creates a problem for the connection pool and for the database.

- **Do not close connections in the finalize method.** As discussed earlier, connections should be opened and closed using the same method. There is, however, a school of thought that promotes closing everything in the `finalize` method, ensuring one location for complete closure. The `finalize` method is not called until the object is garbage-collected, since that is when all finalized methods are called. Closing connections in the `finalize` method can lead to a delay in releasing the connection and should be avoided.

- **If you open a connection, close the connection.** Closing a connection is not absolutely required, since WebSphere Application Server will implicitly close the connection after the connection has timed out. (It has a default value of 30 minutes.) An explicit close, however, expedites the process and follows good application development practices. More specifically, according to the *DB2 UDB/ WebSphere Performance Tuning Guide*, “[I]t is very important that

`ResultSet`, `Statement`, `PreparedStatement`, and `Connection` objects get closed properly in an application. If connections are not closed properly, users may experience long waits for connections to time out, and delay return of the connection to the free pool.”

- **Do not manage data access in container-managed persistence (CMP) beans.** Developers should create CMP beans with the understanding that the container is going to handle the persistence. If one wishes to assume responsibility for persistence, then bean-managed persistence (BMP) should be used. Incorporating BMP processes in CMP beans slows performance.

Prepared Statement Cache

WebSphere Application Server applications may access a database using JDBC via a SQL callable statement, SQL statement call, or SQL prepared statement call. A callable statement removes the need for the SQL compilation process entirely by making a stored procedure call. A statement call is a class that can execute an arbitrary string that is passed to it. The SQL statement is compiled prior to execution, which is a slow process. Applications that repeatedly execute the same SQL statement can decrease processing time by using a prepared statement. A prepared statement redefines a

statement call by separating the compilation process and adding substitution variables. This approach allows the application to prepare the statement once (via compilation) and reuse it at execution multiple times by leveraging different variables.

As a best practice, use prepared statements instead of a SQL statement call for applications that repeatedly execute the same SQL statements.

Session Persistence

Many applications, based on their size and activity level, find the use of memory local session cache on a local application server acceptable for session processing. As the application grows, however, so does its complexity. Such growth may require the incorporation of fault tolerance and redundancy, or the establishment of server clusters. As an application grows, the system administrator may also wish to achieve a higher level of control over the environment. Any or all of these requirements may render in-memory use on a local machine inadequate, requiring the use of session persistence to a database.

In order for data to be persisted to a database, the data must first be serialized. Serialization is the process of writing information to the database or disk, or flattening it out to be trans-



ferred over the wire. To serialize data, you must implement the Java interface `java.io.Serializable`.

Persistent session management does not impact the API. Applications that may require session persistence, therefore, should include `java.io.Serializable` in the code. When the application then scales using session persistence, the code will not require modification.

Once session persistence has been implemented, the session manager will keep the most recent 1,000 sessions in cache memory (this number is configurable) and persist the rest. This process allows the session manager to reduce the number of database accesses required for processing, thereby reducing memory requirements. The decision flow for the session manager is captured in Figure 2.

Following are several best practices to consider when planning for session persistence.

- **Enable session persistence.** Consider making Java objects held by `HttpSession` serializable, even if the application currently operates with local session management. This approach will equip the application should the Web site grow to a size that requires persistence session management. The practice of preparing for growth makes the transition from local to persistent session manage-

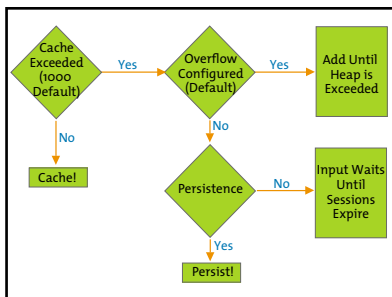


FIG. 2: IN-MEMORY CACHE OVERFLOW ALGORITHM

Source: *DB2 UDB/WebSphere Performance Tuning Guide*

ment transparent to the application.

- **Reduce session size.** Reducing session size becomes particularly important when leveraging persistence sessions. The larger the session, the longer the write to the database, which, in turn, requires more disk i/o. According to *Database System Concepts* (McGraw-Hill), “disk access takes tens of milliseconds, whereas memory access takes a tenth of a microsecond.”

When developing sessions, place easily retrieved information in the application and not in the session. Also, remove stale or old data from the session. According to the *DB2 UDB/WebSphere Performance Tuning Guide*, “[T]he best performance will be realized with session objects that are less than 2K. Once the objects start to exceed 4–5K in size, a significant decrease in performance can be expected.”

- **Release HttpSession when finished.** WebSphere Application Server will release the session once the session has expired, but this can take up to 30 minutes when using the default configuration parameter. Explicitly release the `HttpSession`, forcing an immediate release of memory and garbage collection.

- **Choose persistence options.** IBM offers extensive guidelines for choosing persistence options in the *DB2 UDB/WebSphere Performance Tuning Guide*.

- **Avoid creating HttpSession by JSP by default.** Java Server Pages (JSP), according to the J2EE specification, create `HttpSession` objects by default. In the event you are not going to use the `HttpSession` object, processing requirements would be reduced by not creating the object. To prevent the default `HttpSession` object from being created, add

```
<%@page session="false" %>
```

to the JSP.

- **Tune the cache size.** The default setting provides for 1,000 session objects to be cached. Reducing the number of session objects in turn reduces the amount of required memory for session cache.

- **Add additional application server clones.** To implement vertical scaling, WebSphere Application Server allows the administrator to create multiple instances, or clones, of the application server. This process spreads the requirement for memory across multiple JVMs, thereby reducing the burden of a particular instance. You can also achieve horizontal scaling by adding additional physical hardware resources (servers) to the WebSphere Application Server configuration and implementing WebSphere Application Server clustering. Attempt vertical scaling first to save on purchasing additional hardware, followed by horizontal scaling, with the exception of standard baseline redundancy, which typically supports two physical hardware servers for failover.

- **Tune multirow persistence management.** Multirow session support allows session information obtained from multiple JSPs and servlets to be stored in the session database using multiple rows. For example, Table 1 represents the data collected from session AB12345, depicting information (Session Object data) that a user retrieves.

Referencing Table 1, suppose that the computer needs only a small piece of data, such as first.name, “Michael.” Using multirow persistence, the user can retrieve only the row in the table that houses the first.name variable, leaving the much larger `CandleDemo.String` in the session database until the servlet requests it. However, if the system were leveraging single-row

–continued on page 37

Session ID	Web Application	Property	Small Value	Big Value
AB12345	CandleDemo	CandleDemo.First.Name	“Michael”	
AB12345	CandleDemo	CandleDemo.last.Name	“Pallos”	
AB12345	CandleDemo	CandleDemo.String		A huge string...

TABLE 1: SIMPLIFIED MULTIROW SESSION REPRESENTATION

```

        nextPage = "failure.jsp";
        System.out.println("Response got from the
        Filter");
        //forward request directly to next page instead of
        //Controller Servlet
        rd = req.getRequestDispatcher(nextPage);
        rd.forward(req, resp);
    } else {
        req.setAttribute("login", "loginsuc
        cess");
        //forward request to the Controller Servlet
        //control to next filter or resource
        chain.doFilter(req, resp);
        //Include code to execute on the way back to the
        //client
        System.out.println("Response got from the
        servlet");
    }
} else {
    rd = req.getRequestDispatcher("Welcome.jsp");
    rd.forward(req, resp);
}
}

```

LISTING 2: CONTROLLER.JAVA

```

protected final void doPost(HttpServletRequest request,
    HttpServletResponse response) {

    // begining codes
    //-----User is authenticated
    if (((String)request.getAttribute("login")).equals("login
    success")){
        ArrayList actionreport = new ArrayList();
        actionreport.add("Correct Password");
        session.setAttribute("actionreport",actionreport);
        nextPage="success.jsp";
    }
    if(dispatch){
    RequestDispatcher rd =
    getServletContext().getRequestDispatcher(nextPage);
    }else{ rd.forward(request, response);
}

```

```

session.invalidate();
    }
    // ending codes
}

```

LISTING 3: WEB.XML (FILTER CHAIN AND ORDERING)

```

<filter> First Filter in the chain for Controller
<filter-name>Authentication</filter-name>
  <display-name>Authentication</display-name>
  <filter-class>
com.servlet.filter.Authenticationfilter </filter-class>
</filter>
<filter-mapping>
  <filter-name>Authentication</filter-name>
  <servlet-name>Controller</servlet-name>

< -- <url-pattern>/Controller</url-pattern> -- >
</filter-mapping>
<filter> Second Filter in the chain for Controller
<filter-name>HTTPWrapperfilter</filter-name>
  <display-name> HTTPWrapperfilter </display-name>
  <filter-class>
com.servlet.filter. HTTPWrapperfilter </filter-class>
</filter>
<filter-mapping>
  <filter-name> HTTPWrapperfilter</filter-name>
  <servlet-name>Controller</servlet-name>
</filter-mapping>
<listener>
  <listener-class>
com.servlet.eventlistener.ContextListner</listener-class>
</listener>
<listener>
  <listener-class>
com.servlet.eventlistener.SessionListner</listener-class>
</listener>

```

PERFORMANCE TUNING

—continued from page 30

persistence, it would ask for first name and would also get the huge string since the data is part of the AB12345 session object. When retrieving only selected variables located in the session object, multirow persistence saves time on data retrieval and serialization overhead. Even if multirow session management is incorporated, data contained in the session objects should be kept small, targeting 2K in size and not exceeding 4–5K.

• Tune the session timeout interval.

The WebSphere Application Server default setting for a session time-out is 30 minutes. Depending on the type of Web site being tuned, this number may be too high. At the same time, you must avoid setting the number too low, as it could frustrate users. Ensure that users have ample time to complete online forms. Users who invest a significant amount of time in completing

an online document and then discover their session has timed out when they select Submit may not return to the site or recommend it to others.

Conclusion

The WebSphere Application Server provides an exceptional framework for running distributed Internet applications. Organizations can achieve service-level requirements by optimizing WebSphere software configuration. Best practices serve as a definitive road map for this critical process. Incorporating best practices for connection pooling, prepared statement cache, and session persistence creates an environment in which organizations are empowered to reduce costs associated with hardware and software, while simultaneously enhancing the user experience with faster Internet processing. In the second installment of this series, I will explore strategies for leveraging Enterprise JavaBeans, JDBC, and application monitoring.

References

- Alur, N., Lau, A., Lindquest, S., and Varghese, M. (2002). "WebSphere Application Server and DB2 UDB Performance." *DB2 UDB/ WebSphere Performance Tuning Guide*. IBM Redbooks.
- Erickson, D., Lauzon, S., and Modjeski, M. (2001, August). *WebSphere Connection Pooling: www-3.ibm.com/software/web servers/appserv/whitepapers/connection_pool.pdf*
- Silberschatz, A., Korth, H., and Sudarshan, S. (2002). *Database System Concepts (4th ed.)*. McGraw-Hill Higher Education.
- *Java 2 Platform, Standard Edition, v1.3.1 API Specifications*: <http://java.sun.com/j2se/1.3/docs/api/>
- *JDBC Data Access API, JDBC 2.0 Optional Package API*: <http://java.sun.com/products/jdbc/index.html>
- *JDBC Data Access API*: [http://indus. try.java.sun.com/products/jdbc/ drivers](http://indus. try.java.sun.com/products/jdbc/drivers) 