

Part 2: EJBs, JDBC, and application development and monitoring

WebSphere and Database Performance Tuning

BY MICHAEL S. PALLOS

In a business environment defined by requirements to elevate service levels while reducing IT operational costs, developers seek proven strategies to optimize the production runtime environment of their WebSphere Application Server implementations. The nearly universal objective of IT leaders today is to improve application performance and maximize the environment's ability to support broad business objectives.

In the May issue of *WebSphere Developer's Journal* [Vol. 2, issue 5], I presented the first of a two-part series that provides strategies for tuning network and database interfaces to optimize IBM WebSphere Application Server implementations. That article discussed best practices for database connection pooling, prepared statement cache, and session persistence. In this article, I present best practices and tuning techniques for EJBs (Enterprise JavaBeans), JDBC (Java Database Connectivity), and application development and monitoring.

Enterprise JavaBeans

The EJB specification is the foundation for Java 2 Enterprise Edition (J2EE), offering component services such as distributed transactions,

security, and life-cycle management. There are two types of EJBs – session beans and entity beans (J2EE 1.3 also includes message-driven beans). A session bean instance belongs to a specific user and maintains the user's state as he or she interacts with the Web site. User status information, however, is lost in the event of a catastrophic system failure. A shopping cart utilized by an online shopper exemplifies the use of a session bean. As the user progresses through the Web site, he or she can add and remove items from the cart – the session bean. Once the user makes a purchase or chooses to leave the Web site without completing a purchase, the user's interaction with the site is terminated and the session bean is destroyed.

An entity bean represents data – typically a row in a database – and

can be shared among multiple users. Continuing with the shopping cart example, if the user makes a purchase, the data will persist to a database via an entity bean. Should the transaction require the creation of an invoice, an entity bean would be created containing the user's data. Because it is persisted to the database, entity bean data will survive a system crash.

EJBs offer many advantages, including the ability to leverage distributed objects and transactional services. These advantages, however, come at a price – increased complexity. EJBs that are architected or incorporated incorrectly can have an adverse impact on application processing. Proper EJB utilization and optimization, conversely, reduces processing cycles, which, in turn increases application performance and enhances the end-user experience.

EJB best practices that optimize the storage and retrieval of persistence data include the following:

- **Use the appropriate isolation level.** Isolation levels are used to restrict access to a resource to which other concurrent transactions have access. They allow users to lock down or isolate shared database resources to four levels of granularity:

–*Read Uncommitted:* The transaction can read uncommitted data from other transactions.

Uncommitted transactions yield the lowest overhead since the container is doing the least amount of work. Imagine, for instance, that a user reads the database and retrieves certain values. Another user then starts a BEGIN WORK transaction and inserts values into the database. If the first user performs another database read before the second user has executed a COMMIT WORK or ROLLBACK command, he or she will read the uncommit-

ted data that has been temporarily inserted into the database. The scenario just described is also known as a “dirty read.”

–*Read Committed*: The transaction can read committed data only. If we repeat the scenario described above with a transaction state set to Read Committed, a user performing a read on a database containing data from a BEGIN WORK that has not been COMMITTED will not be able to access the uncommitted data. The user can only read committed data. A Read Committed is more restrictive than a Read Uncommitted isolation level, requiring the system to work harder. A Read Committed status, therefore, has a greater impact on processing than a Read Uncommitted status.

–*Repeatable Read*: The transaction is guaranteed to read back the same data on each successive read. A Repeatable Read is more restrictive than a Read Committed, having an even greater impact on processing.

–*Serializable*: All transactions are serialized or completely isolated from each other. A Serializable transaction is the most restrictive isolation level, and has the greatest impact on processing. An EJB that uses a serialized transaction isolation level is guaranteed to achieve consistent results from the database since the database is locked from other users, essentially creating single-threaded processing. While rookie developers are often tempted to incorporate the serializable isolation level to ensure data integrity, it has major constraints. Although results are guaranteed, all concurrent users are locked out of the database, which slows down the application. To optimize performance for data access, developers must fully understand the difference between the four isolation levels and their indicated uses.

- **Carefully define access intent.** The access intent attribute contains one of two states: (1) read and (2)

update, with the default setting being update. For methods that are going to be READ ONLY, setting access intent to READ will increase data access performance. Under such conditions, the database is accessed with an intent to read – and not update – thus removing unnecessary database locking.

Java Database Connectivity

JDBC provides a standard library for accessing relational data, allowing users to develop SQL calls using the Java Application Programming Interface (API). The JDBC driver is responsible for the data access communication interface with the database management system (DBMS).

Selecting the appropriate driver is fundamental to improving performance. According to Sun Microsystems, there are four categories of JDBC drivers (Types 1–4), and more than 177 models.

- **Type 1 – JDBC–Open Database Connectivity (ODBC) bridge:** A Type 1 driver provides JDBC access to one or more ODBC drivers. Type 1 drivers assist companies that already possess a large ODBC population with the JDBC educational process. Type 1 drivers are slow, however, because they require JDBC-ODBC translation. As such, they are not well suited for large enterprises.
- **Type 2 – Partial Java driver:** A Type 2 driver converts the calls made from the JDBC API to the receiving machine’s API for a specific database (DB2, Oracle, Sybase, SQL Server, etc.). A Type 2 driver contains compiled code for the back-end system. Type 2 drivers process more quickly than Type 1 drivers. The code must be compiled, however, for every operating system on which the application runs. In Windows NT and z/OS development, organizations may wish to develop with a Type 4 driver and then transition to a Type 2 driver for production.
- **Type 3 – Pure Java driver for database middleware:** A Type 3 driver provides connectivity to

many different databases, translating JDBC calls into the middleware vendor’s protocol and then into the database-specific protocol via the middleware server. A Type 3 driver is often faster than Type 1 and 2 drivers, and is useful if an organization wishes to connect to multiple database types. Database-specific code, however, must reside on the middle tier. If the application is going to run on different operating systems, a Type 4 driver may be more appropriate.

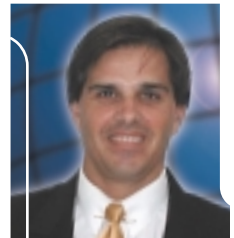
- **Type 4 – Direct-to-database Java driver:** A Type 4 driver converts JDBC calls into packets that are transferred over the network in the database’s proprietary format, allowing a direct call from the client to the database without a middle tier. Type 4 drivers often offer better performance than Type 1 or 2 drivers; do not require additional code on client or server machines; and can be downloaded dynamically. Type 4 drivers, however, are not optimized for the operating system and are unable to take advantage of operating system features. Users also need a different driver for each different database.

When developing on multiple platforms, many aspects of testing can be simplified by using a Type 4 driver. A Type 2 driver can be incorporated after system testing and production is completed. According to Sun Microsystems and IBM, Type 2 JDBC drivers offer the best performance when incorporating distributed transactions and catalog databases.

When selecting a driver, developers should also consider the JDBC API levels that it supports. Currently, there are two JDBC API levels. While JDBC 1.0 is the default, WebSphere requires JDBC 2.0. The JDBC API best practice is to incorporate version 2.0.

Application Development and Monitoring

WebSphere Application Server development and monitoring tools



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

can offer insight into potential application bottleneck areas that require additional attention to achieve optimal performance. An ideal development tool identifies workflow analysis down to the method and thread level, offering real-time and historical information. Monitoring tools should also be able to report results on all elements contained within the application. These elements include the end user's perspective through WebSphere Application Server, DBMS, transport layers, connection layer, and, if incorporated, any legacy system components.

Three best practices to consider when selecting and/or deploying monitoring capabilities follow:

- **Select a development tool that delivers granular, real-time information.** During development, a tool can help to identify and/or eliminate memory leaks, bottlenecks, and optimization. Development tools should offer granularity to the method level and thread level, SQL calls, and heap insight. The tool should also provide the ability to look into the application during execution and deliver sufficient analysis to allow developers to proactively address problems.
- **Select monitoring tools that assist with the early stages of the development phase.** Monitoring tools should be capable of capturing the complete end-to-end scenario, including the end-user experience, latency time, and the performance and availability of

systems external to WebSphere Application Server, such as CICS, DB2, WebSphere MQ, and B2B. Pure Java monitoring tools are excellent for initial development. However, many WebSphere Application Server applications incorporate external Java components to complete application processing. A desirable monitoring tool not only reports on the Java pieces of the application, but the entire application portfolio. If a highly productive Java application is executing on a machine that is underutilized, the operating system monitoring agent can then report the problem to the Java developer or operations personnel.

- **Select a solution that offers both development and monitoring tools.** (One choice could be the Candle PathWAI solution suite.) This strategy simplifies training, development, and production rollout.

Conclusion

Today's developers face unprecedented pressure to create WebSphere applications that roll out rapidly and flawlessly, and adhere to business service-level requirements for processing and end-user experience. Once deployed, the challenges begin anew, as developers, system administrators, and operations work to optimize the production runtime environment for their WebSphere Application Server implementations. Best practices for tuning

WebSphere Application Server persistence layers, performance, and database interfaces represent the most direct and hazard-free path to optimizing WebSphere Application Server implementations.

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.
- Endrei, M., Cluning, R., Daomanee, W., Heyward, J., Iyengar, A., Mauny, I., et al. (2002). *IBM WebSphere V4.0 Advanced Edition Handbook*. IBM.
- Erickson, D., Lauzon, S., and Modjeski, M. (2001, August). *WebSphere Connection Pooling: www-3.ibm.com/software/web_servers/appserv/whitepapers/connection_pool.pdf*
- Hutchison, G. (2002). *DB2/WebSphere Integration: www.websphere-users.ca/presentations/hutchison.PDF*
- Monson-Haefel, R. (2000). *Enterprise JavaBeans*. O'Reilly & Associates.
- Silberschatz, A., Korth, H., and Sudarshan, S. (2002). *Database System Concepts*. 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 Drivers: <http://industry.java.sun.com/products/jdbc/drivers>* 

"WebSphere Application Server development and monitoring tools can offer insight into potential application bottleneck areas that require additional attention to achieve optimal performance"