

Best Practices & Trade-offs for Efficient, Reliable Java Persistence

Kuassi Mensah
Oracle Corporation
USA

Keywords:

JDBC, Universal Connection Pool, RAC, Data Guard, Fast Connection Failover, Runtime Load Balancing, Web-Affinity, Transaction-Affinity

Introduction

All Java persistence solutions are built on top of a foundation: JDBC. Your Java applications are doomed to inefficiency if you fail JDBC. In this technical session, you will learn from Oracle development, how Oracle JDBC and the Universal Connection Pool allow: implementing a reliable and highly-available Java Persistence. Coverage includes how to: optimize LOB operations and bulk data transfers; implement cache invalidation with Change Notification; support SSL, Kerberos and Radius authentication; trade speed for space or vice versa; scale and fail-over connections with Fast Connection Failover and Runtime Load balancing; implement Web-session and transaction affinity in RAC and Data Guard environments. Finally the session will discuss how to debug and troubleshoot common Java/JDBC applications issues.

How to Optimize LOB operations and Bulk Data Transfer

LOB Prefetch

LOB Prefetch allows retrieving the LOB locator and LOB data in the same roundtrip. LOB prefetch is extremely effective for small LOBs (less than 5k) but its effect diminishes as the LOB size increases. As shown in Illustration 1, for LOBs that are 1K in size, the numbers show a throughput of 1652 LOB/sec with LOB prefetch turned ON versus 75 LOB/sec without LOB prefetch turned on. For LOBs that are 10K in size, the difference is not as significant however there is a difference (108 LOB/sec with LOB prefetch turned on versus 53 LOB/sec with LOB prefetch turned off).

SecureFile LOB

LOBs with STORE AS SECUREFILE option reduce storage and increase access performance. Oracle JDBC supports the new functionalities of SecureFile LOBs through new APIs including:

```
java.sql.{ClobShare|BlobShare}  
oracle.sql.{BlobRegion|ClobRegion}  
oracle.sql.{setOptions|getOptions}  
oracle.sql.{Blob|Clob}.{setSharedRegions|getSharedRegions}
```

Traditionally, LOB data is copied into an internal server buffer before streaming it to client. In Oracle Database 11g Release 2, Net Services furnish “Zero-copy IO” interface which allows bypassing the copying. Oracle JDBC 11g R2 leverages Net Services “Zero-copy IO”; as a result SecureFiles LOB operations are faster.

```
setupSecureFile(); Blob.getBytes();
```

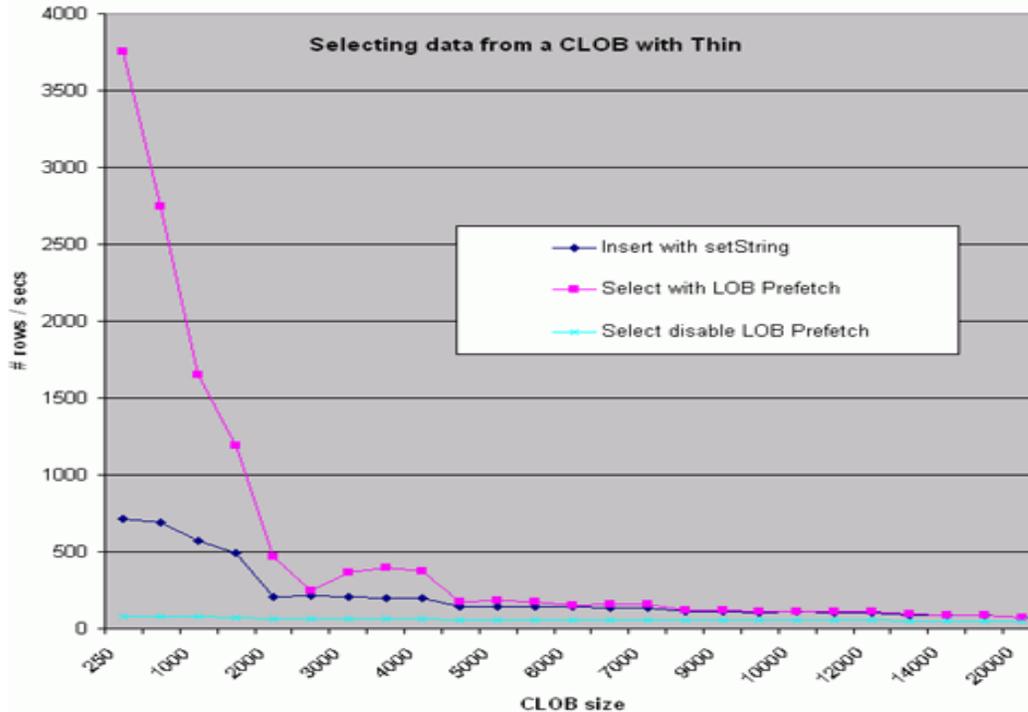


Illustration. 1: LOB Prefetch Small vs Large LOBs

Tuning Net Services for Bulk Data Transfer

Net Services parameter Session Data Unit (SDU) controls Net Services packet size. The default value is 2K in pre-11g database then 8K in 11g. For Bulk data transfer, it can be set to 32K in JDBC URL, or SQLNET.ORA (DEFAULT_SDU_ZIE) or TNSNAMES.ORA (SDU in ADDRESS).

Larger SDU gives better network throughput, fewer system calls to send and receive data, and less CPU consumption.

How to implement Cache invalidation with Change Notification

JDBC support for Query Notification is a scalability mechanism, which let's a JDBC thread subscribe to notification of changes in a query result set so as to invalidate cached data in client or middle tiers. Using the feature involves the following steps:

- (i) Registration: JDBC applications can register SQL queries (i.e., create registrations) using either the JDBC-style or PL/SQL-style registration.
- (ii) SQL Query Association (Table Changes): associate SQL queries with the registrations.
- (iii) RDBMS Notification: upon DDL and/or DML changes that impact the registered queries or their result sets, the Oracle database notifies the JDBC thread through a dedicated network connection. The driver transforms these notifications into Java events, received by the application.

Middle-tier may use this feature to invalidate and refresh data caches, timely. See more details on the APIs in the *Oracle Database 11g JDBC doc*.

JDBC Memory Management and How to Trade Space for Speed (and vice versa)

JDBC Memory Allocation

JDBC memory management is controlled by (i) table/column definition, (ii) the query syntax, and (iii) the `fetchSize`.

- a) Defining a column as `VARCHAR2(4000)` column requires 8K bytes per row while a `VARCHAR2(20)` column needs only 40 bytes per row. If the column in fact never holds more than 20 characters, then most of the buffer space allocated by the driver for a `VARCHAR2(4000)` column is wasted. Similarly
- b) Querying and returning unnecessary columns i.e., `SELECT * from mytable` instead of `SELECT col1, col2, col3 from mytable` will lead to the allocation of unnecessary large buffer(s) and impacts performance
- c) A query returning 200 `VARCHAR(4000)` columns with a `fetchSize` of 100 will allocate $(2 * 4000) * 200 * 100 = 160\text{MB}$; tuning `fetchSize` can have a big performance impact.

Trading Space for Speed (and vice versa)¹

Zeroing `defineBytes` and `defineChars` array buffers is required by Java lang. spec. but is expensive. Solution: trading space for speed or vice versa.

Trading Space for Speed: Statement Caching

- reuses buffers
- eliminates the overhead of repeated cursor creation
- prevents repeated statement parsing and creation

Oracle JDBC Drivers furnish two Statement Caching: Implicit statement caching and Explicit statement caching.

Trading Speed for Space: JDBC Memory Management

- i) In 10.2, setting `oracle.jdbc.freeMemoryOnEnterImplicitCache` connection property results in releasing buffers when the Statement object is returned to the cache. If the application has many open statements at once, this can be a problem.
- ii) In 11.1.0.7 and up, setting `oracle.jdbc.maxCachedBufferSize` connection property mitigates this problem. This property bounds the maximum size of buffer that will be saved in the buffer cache. All larger buffers are freed when the `PreparedStatement` is put in the Implicit Statement Cache and reallocated when the `PreparedStatement` is retrieved from the cache.
- iii) In 11.2 JDBC furnishes a smarter and more sophisticated memory allocation. This buffer cache has multiple buckets. All buffers in a bucket are of the same size and that size is predetermined. When a `PreparedStatement` is executed for the first time the driver gets a buffer from the bucket holding the smallest size buffers that will hold the result. If there is no buffer in the bucket, the driver allocates a new buffer of the predefined size corresponding to the bucket. When a `PreparedStatement` is closed, the buffers are returned to their appropriate buckets. Since buffers are used for a range of size requirements, the buffers are usually somewhat larger than the minimum required. The discrepancy is limited and in practice has no impact.

Recommendation: use the very latest JDBC drivers as soon as possible.

¹ See the complete white paper @ <http://www.oracle.com/technetwork/database/enterprise-edition/memory.pdf>

Universal Connection Pool (UCP)

A single Java Connection Pool which supports any type of Java connection including JDBC, XA, JCA, and LDAP. UCP has a seamless integration with Oracle Database RAC and Data Guard; may support any Database (Oracle, non-Oracle); and may support any Java Application Server (Oracle, non-Oracle).

UCP furnishes the following connection services in Oracle Database HA configurations.

- Fast Connection Failover (FCF)
- Runtime Connection Load Balancing (RCLB)
- Web Session Based Affinity to RAC instance
- Transaction Based Affinity to RAC instance

Oracle Database HA Configurations

The Oracle Database furnishes the following High-Availability configurations:

- Single Instance HA
- Cold Failover Cluster
- RAC, RAC One, RAC with Vendor Clusterware
- Data Guard Physical Standby (single Instance or RAC with/without Broker)
- Data Guard Logical Standby (Single Instance or RAC with/without Broker)

A detailed coverage of each of these configurations can be found in Oracle Database 11g Release 2 High Availability Overview document @

http://download.oracle.com/docs/cd/E14072_01/server.112/e10804.pdf

Fast Application Notification (FAN)

FAN allows emitting events when a managed service, instance or site² goes up or down. The events are then propagated by a notification mechanism; FAN uses either Oracle Notification System (ONS) to send the events to JDBC clients when conditions change. The main benefits of FAN when compared to TCP timeouts are: fast detection of condition change and fast notification.

Fast Connection Failover (FCF)

FCF is a process that Oracle JDBC uses for managing FAN events. The key features of FCF are:

- a) Rapid database service/instance/node failure detection then abort and removal of invalid connections from the pool
- b) UCP can be configured for Fast Connection Failover (FCF) to automatically connect to a new primary database upon failover.

The following code fragment shows how a plain JDBC application would enable Fast Connection Failover and retry to get a new connection upon instance failure.

² After a failover, the Oracle Data Guard Broker (broker) publishes FAN events.

```

PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
pds.setONSConfiguration("nodes=racnode1:4200,racnode2:4200");
pds.setFastConnectionFailoverEnabled(true);
.....
boolean retry = false;
do {
    try {
        Connection conn = pds.getConnection("scott", "tiger");
        // Data operations using conn object

        retry = false;
    } catch (SQLException sqlexc) {
        if (conn == null || !((ValidConnection)conn).isValid())
            retry = true;
    }
} while (retry);
.....

```

Runtime Connection Load Balancing (RCLB)

Oracle RAC emits Load Balancing Advisories that allow implementing Runtime Connection Load Balancing.

RCLB: what for?

- Manages pooled connections for high performance and scalability
- Receives continuous recommendations on the percentage of work to route to Database instances
- Adjusts distribution of work based on different backend node capacities such as CPU capacity or response time
- Reacts quickly to changes in Cluster Reconfiguration, Application workload, overworked nodes or hangs

Enabling RCLB

- Enable Fast Connection Failover
- Set RAC load balancing advisory GOAL for each service, using DBMS_SERVICE
 - NONE (default)
 - GOAL_SERVICE_TIME – best service time overall
 DBMS_SERVICE.CREATE_SERVICE('SERVICE1','SERVICE1.company.com',
 goal => DBMS_SERVICE.GOAL_SERVICE_TIME)
 - GOAL_THROUGHPUT -- best throughput overall
 DBMS_SERVICE.CREATE_SERVICE ('SERVICE2','SERVICE2.company.com',
 goal => DBMS_SERVICE.GOAL_THROUGHPUT)

Alternatively, set service goal using the init.ora goal parameter
- Set CLB_GOAL for the listener to CLB_GOAL_SHORT
 - CLB_GOAL_SHORT – connection load balancing uses RAC Load Balancing Advisory
 DBMS_SERVICE.MODIFY_SERVICE ('SERVICE1','SERVICE1.company.com',
 clb_goal => DBMS_SERVICE.CLB_GOAL_SHORT)
 - CLB_GOAL_LONG – for applications that have long-lived connections

```
DBMS_SERVICE.MODIFY_SERVICE ('SERVICE2', 'SERVICE2.company.com',  
                              clb_goal => DBMS_SERVICE.CLB_GOAL_LONG)
```

Alternatively, set connection load balancing goal using srvctl

Web-Session Affinity

What for?

Benefits applications with repeated operations against the same client records such as:

- 1) Online shopping: create a cart, add to the cart, modify the cart, display the cart, add to the cart, display the cart, etc.
- 2) Online banking: logon to an account, display transactions, enter bill line items, confirm payment transaction, display pending payments, verify a vendor's history, etc.
- 3) Self-service human resources: logon to an account, display paystub, check vacation balance, enter vacation time, etc.

How does it work?

Affinity scope: determined by the web-session life cycle, affinity hint, or available connections in the pool

Affinity hints: each RAC instance evaluates performance metrics and sends affinity hints per Service to UCP

Affinity Contexts: stored within application and accessed via Affinity Callback

Affinity Callback: used by the pool to store and retrieve Affinity Contexts

- The first connection request uses RCLB to select a connection
- Subsequent requests enforce Affinity
- Connection selection falls back to RCLB after Affinity ends
- Affinity is only a hint: UCP resorts to RCLB whenever a desired connection is not found

Code Snippet

```
PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();  
pds.setONSConfiguration("nodes=racnode1:4200,racnode2:4200");  
pds.setFastConnectionFailoverEnabled(true);  
ConnectionAffinityCallback cbk = new MyCallback();  
Pds.registerConnectionAffinityCallback(cbk);  
.....  
class MyCallback implements ConnectionAffinityCallback  
{  
    .....  
    Object appAffinityContext = null;  
    AffinityPolicy policy = AffinityPolicy.WEBSESSION_AFFINITY;  
    public boolean setConnectionAffinityContext(Object cxt)  
    { appAffinityContext = cxt; }  
    public Object getConnectionAffinityContext()  
    { return appAffinityContext; }  
    .....  
}
```

Transaction Based Affinity

What for?

- a) Enables XA and RAC to work together with optimal performance – Eliminates current single DTP service limitation for XA/RAC
- b) Transaction affinity is the ability to automatically localize a global transaction to a single RAC instance
- c) Transaction Affinity scope is the life of a global transaction
 - first connection request for a global transaction uses RCLB
 - subsequent requests uses affinity and are routed to the same RAC instance when XA first started

How does it work?

Affinity scope: determined by the transaction lifecycle and database failure events

Affinity contexts: stored within application and accessed via Affinity Callback; cleared by application after transaction completes; Can be propagated with transaction contexts

Affinity Callback: used by the pool to store and retrieve Affinity Contexts

```
PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
pds.setFastConnectionFailoverEnabled(true);
ConnectionAffinityCallback cbk = new MyCallback();
Pds.registerConnectionAffinityCallback(cbk);
.....
class MyCallback implements ConnectionAffinityCallback
{
    .....
    Object appAffinityContext = null;
    AffinityPolicy policy = AffinityPolicy.TRANSACTION_AFFINITY;
    public boolean setConnectionAffinityContext(Object cxt)
    { appAffinityContext = cxt; }
    public Object getConnectionAffinityContext()
    { return appAffinityContext; }
    .....
}
```

Tips for Debugging Common Issues

- a) *Determine which JDBC JAR file is being used?*

```
System.out.println("jar file: "
    +oracle.jdbc.OracleDriver.class.getResource(""));
```

- b) *Which JDBC driver version is in use?*

Starting with 11g drivers

```
java -jar ojdbc5.jar -version
```

Starting with 10g drivers

```
Unzip -c ojdbc14.jar META_INF/MANIFEST.MF | grep -i version
```

c) *Why i am getting* “java.lang.SecurityException: sealing violation: package oracle.jdbc.driver is sealed“?

Probably because you have multiple JDBC JAR files in your CLASSPATH.

d) *What Could Slow Down JDBC Applications?*

- Check SDU setting in Connect String or in Net Services (a.k.a. SQLNET) trace and tune appropriately
- Avoid frequent Connect-Disconnect: look at AWR/ADDM reports or Listener log files, then enable Connection Caching (UCP)
- Avoid waiting for unreachable host
Set connection timeout properties `oracle.net.CONNECT_TIMEOUT`,
`DriverManager.setLoginTimeout(int)`
- Ensure that the server is still reachable i.e., **sqlplus user/passwd**
- Enable keep-alive to detect broken connection faster
(**ENABLE=BROKEN**) in the connect string
Set `tcp_keepalive_time`, `tcp_keepalive_intvl`

e) *Optimizing Statements Performance*

Turn on Statement Caching

- Enable bind tracing (`ALTER SESSION SET EVENTS '10046 trace name context forever, level 8';`)
- Use tkprof to see the number of parse, execute & fetch; if more than 1 parse then enable statement cache

Monitor Fetch-size and registerOutParameter size

- Enable `java.util logging`³
The parameter values would be displayed in the generated trace file
- Use dynamic binding instead of static bindings
Analyze statements being executed by JDBC logging
`oracle.jdbc.driver.level = CONFIG`
Change the SQL to dynamic binding to avoid SQL Injection issues and also to achieve better performance
- Use batching for repeated DML
Analyze statement being executed by JDBC logging
`oracle.jdbc.driver.level = CONFIG`
Enable batching for similar statement execution

Contact address:

Kuassi Mensah

Oracle Corporation

400 Oracle Parkway - 94065, Redwood City, USA

Phone: (+1) 650 607 2229

Fax: (+1) 650 506 7225

Email kuassi.mensah@oracle.com

Blog : <http://db360.blogspot.com>

³ http://www.oracle.com/technology/tech/java/sqlj_jdbc/pdf/11.2%20logging%20white%20paper.pdf