An Oracle White Paper
June 2013

Java Performance, Scalability, Availability & Security with Oracle Database 12c

1 - Introduction

No matter your programming model Java SE, Java EE, JPA/POJO, the RDBMS has a significant impact on the efficiency[1] of your Java components, applications, frameworks.
Have you ever needed to deploy Java applications with tens of thousands of concurrent users?
Have you ever experienced paying twice the same flight ticket, the same article or your taxes?
Have you ever wanted the system to just deal with database failure and not ask you to restart your transaction from start? Looking to exploit the new Oracle Multitenant Container Database with Java?
If you are a Java architect, Java designer or wannabe looking to exploit new Oracle database 12c enhancements in the areas of performance, scalability, availability, security and manageability/ease-of-use, this is the paper for you.
The Oracle database furnishes the JDBC drivers, the Universal Connection Pool (UCP), the database resident JVM, the JPublisher utility and the SQLJ driver[2]; this paper covers the new Oracle database 12c features across these components.

[1] Including performance, scalability, availability manageability and security

[2] The SQLJ driver is not covered in this paper.

2 – Support for Latest Java Standards

Through its Java components[3], the Oracle database 12c supports Java SE 7 (in addition to Java SE 6), JDBC 4.1, in-database JNDI, and in-database Java Logging.

Java SE 7 and Multiple Java SE

Both Oracle JDBC and UCP now support Java SE 7 (`ojdbc7.jar, ucp.jar).`

The database-resident Java VM which allows running plain Java SE applications directly in the database, now supports Java SE 6 (the default) and Java SE 7, with the ability to upgrade/downgrade from one to the other (see the Java Developer guide for more details).

JDBC 4.1

Oracle Database 12c now supports the JDBC 4.1 `getObject()` method with two signatures.

```
getObject(int parameterIndex,java.lang.Class<T> type)
            throws SQLException

getObject(java.lang.String parameterName,java.lang.Class<T> type)
            throws SQLException
```

Example

```
ResultSet rs = . . . ;
Character c = rs.getObject(1, java.lang.Character.class);
```

Other JDBC 4.1 feature include `try-with-resources` however, Oracle JDBC does not yet have support for `setObject` and the `RowSetFactory` interface.

In-Database JNDI

The database resident Java VM now supports a general purpose directory service for storing/binding objects references including:

- A namespace: similar to Unix file system structure
  /pub
  /etc
- A namespace browser (`ojvmjava` utility): enables browsing permissions and properties of objects stored in the OJDS

```
ojvmjava -u scott/tiger
connected to scott
--OJVMJAVA--
--type "help" at the command line for help message
$ ls
etc/      public/
```

[3] JDBC drivers, UCP, and embedded Java VM

```
$ bindds ds1 -host localhost -port 5521 -sid jvmj4 -user scott
$ ls
ds1

$ bindurl myURL http://www.oracle.com -g PUBLIC
$ ls
myURL
```

- A Java Directory Service Namespace Provider (OJDS)

```
String url = "ojds://thin:"+host+":"+port+":"+sid;
DirContext cctx = (DirContext)ctx.lookup(url+"/public");
System.out.println("'/public' is bound to "+cctx);
Attributes attrs = new BasicAttributes(true);
attrs.put("owner","xxx");
attrs.put("read","xxx");
attrs.put("write","xxx");
attrs.put("execute","xxx");
ctx.bind(url+"/public/myObj",new Byte("100"),attrs);
System.out.println("'/public/myObj' is bound to
                                "+ctx.lookup(url+"/public/myObj"));
Attributes answer = ctx.getAttributes(url+"/public/myObj");
```

In-Database Java Logging

The database-resident Java VM now supports Java Logging and furnishes a `logging.properties` file. It is initialized once per session with the `LogManager` API, extended with the database resource lookup.

- If `$ORACLE_HOME/javavm/lib/logging.properties` resource is set in the current user schema, this resource is used for configuring the `LogManager` and the `java.util.logging.config.file` property is set
- Otherwise the `$ORACLE_HOME/javavm/lib/logging.properties` resource in the `SYS` schema is used

You may configure a different properties file and load them in your schema using the `loadjava` utility.

3 – Support for the Multitenant Container Database

The Oracle database 12c introduces a new multitenant architecture consisting of a root infrastructure called CDB which contains exclusively Oracle provided metadata, then a set of pluggable databases (PDBs), which are full-fledged databases containing customers and applications data and metadata.
The benefits of multitenant architecture are:

- Fast provisioning of a new database or of a copy of an existing database.
- Fast redeployment, by unplug and plug, of an existing database to a new platform.
- Quickly patch or upgrade the Oracle Database version for many databases and for the cost of doing it once.
- Patch or upgrade by unplugging a PDB and plugging it into a different container database (CDB) in a later version.
- A server machine sustains more PDB databases than traditional non-CDB databases.
- Separation of roles and duties of CDB administrator from a PDB administrator.

From Java containers, frameworks and applications perspective, the PDBs feel and operate identically to non-CDB databases i.e., traditional Oracle databases provided the "*service name*" (Oracle Net Services parlance) of the target PDB is used in JDBC URL connect strings.

Multitenant DataSource for Java

In order to allow sharing a single pool of connections across multiple PDBs i.e., tenants, Oracle JDBC and UCP furnish the Multitenant Datasource for Java. In this release, it is based on a combination of (i) global database user i.e., may access any PDB, (ii) UCP connection labeling, (iii) the new "`ALTER SESSION SET CONTAINER=<PDB Name>`", and (iv) the UCP connection labeling callback interface.

Here is a sketch of how Multitenant Datasource for Java works[4]

1. Tenant1 asks for a connection to PDB1, by calling `getConnection()` with the corresponding label (mapped to database id)
2. UCP searches the pool for a free connection (tentatively with the specified label)
3. If Conn1 label reads "PDB1" then it is handed to JDBC then to Tenant1 for use.
4. Otherwise
   (i) the user-implemented callback "`configure`" method is invoked by UCP to set Conn1 to PDB1 using `ALTER SESSION SET CONTAINER …`
   (ii) the " … `SET CONTAINER` …" statement is passed to the server and parsed
   (iii) the server executes the statement, assigns Tetant1 the PDB1-specific-role[5] and connects Conn1 to PDB1 then returns the corresponding database id ("dbid") and other properties to JDBC
   (iv) JDBC notifies the conn pool (UCP), then hands Conn1 to Tenant1 for use.

3 – JDBC Support for New SQL Data Types

Java applications may leverage new data types including: 32K VARCHAR, NVARCHAR, and RAW, invisible/hidden columns, implicit results, auto-increment or IDENTITY columns, PL/SQL packaged types and as parameters, larger row count data type, and XStream enhancements.

32K VARCHAR, NVARCHAR, and RAW

The maximum size of the `VARCHAR2, NVARCHAR2, and RAW` data types has been increased from 4,000 to 32,767 bytes. Java applications using JDBC will no longer need to switch to large objects (LOBs) for data inferior to 32K in size; indexes may also be built on top of columns declared with such data types.
In order to use these extended data types, database administrators must perform the following steps

[4] A forthcoming Multitenant Datasource for Java white paper will be posted @ http://www.oracle.com/technetwork/database/application-development/index-099369.html

[5] Once in a PDB a global user cannot do much without being assigned a password protected role.

1. Set the COMPATIBLE initialization parameter to 12.0.0.0.
2. Set the MAX_STRING_SIZE initialization parameter to EXTENDED.
3. Run the rdbms/admin/utl32k.sql script.

Invisible or Hidden Columns

New columns can be created or added to table(s) then hidden, using the INVISIBLE SQL keyword. Invisible columns are not displayed during generic access such as a "SELECT * FROM table" or "DESCRIBE table" however these may be displayed when specified explicitly in the SELECT list.
For Java, Oracle JDBC furnishes metadata and a method (isColumnInvisible) to check whether a column is invisible / hidden or not.

Example

```
OracleResultSetMetaData rsmd = (OracleResultSetMetaData)rset.getMetaData();
...
System.out.println("Visibility:" + rsmd.isColumnInvisible(2));
```

Implicit Results

Problem to solve: make easy the migration of Java applications from foreign databases to Oracle. Implicit Results allows Java applications to retrieve the return of stored procedures (PL/SQL, Java) directly, using the PL/SQL procedure DBMS_SQL.RETURN_RESULT, without the need to use Ref Cursor.

```
create or replace procedure p_imres as
result sys_refcursor;
begin
            open result for select * from tab;
            dbms_sql.return_result(result);
end;
```

Java applications may use the following methods to consume implicit results

- getMoreResults() or getMoreResults(int) to check if there are more results available in the result set.
  The int parameter that can have one of the following values:
  - KEEP_CURRENT_RESULT
  - CLOSE_ALL_RESULTS
  - CLOSE_CURRENT_RESULT
- getResultSet: iteratively retrieves each implicit result from an executed PL/SQL statement.

Example

```
CallableStatement cstmt = null;
ResultSet rs = null;
cstmt = conn.prepareCall("{call p_imres()}");
cstmt.execute();
boolean resultsAvailable = cstmt.getMoreResults();
```

Auto-Increment Columns (IDENTITY columns)

Problem to solve: portability or easy migration of Java, C, C++, or C# applications built on foreign RDBMS.
The Oracle database 12*c* implements the ANSI compliant and automatically incrementing columns using the SQL keyword `IDENTITY`.

```
CREATE TABLE t1 (c1 NUMBER GENERATED BY DEFAULT ON NULL AS IDENTITY, c2
VARCHAR2(10));
```

PL/SQL Package Types as parameters

For Java application, Oracle JDBC now supports PL/SQL package types as parameters using ADT APIs i.e., STRUCT or ARRAY classes or a custom Java class

- Use `%ROWTYPE` to create PLSQL package types, then obtain the rows of a table as `java.sql.Array` or `java.sql.Struct`
- The PL/SQL package type names must be specified as `<schema name>.<package name>.<type name>` or `<package name>.<type name>`

This feature makes easier to fetch query results as ADT, thereby simplifying Object-Relational mapping.

PLSQL Package type example

```
CREATE OR REPLACE PACKAGE pack1 AS
  TYPE employee_rowtype_array IS A TABLE OF Employee%ROWTYPE;
END;
/
```

Java/JDBC Code

```
CallableStatement cstmt =
     connection.prepareCall("BEGIN SELECT * INTO :1 FROM EMPLOYEE;
END;");
cstmt.registerOutParameter(1, OracleTypes.ARRAY,
  "PACK1.EMPLOYEE_ROWTYPE_ARRAY");
cstmt.execute();
Array employeeArray = cstmt.getArray(1);
```

Schema objects such as `INTEGER, VARCHAR` are mapped respectively to Java `int and String`.

5 – Java Performance and Scalability

New performance and scalability enhancements include: new JDBC memory management Database Resident Connection Pool, very large network buffers (SDU), and Runtime Connections Load Balancing across geographies (Global Data Services).

New JDBC Memory Management

JDBC memory management has a significant impact on the JVM memory and ultimately on Java performance.
Problem to solve: given a query returning 1000 rows (fetchsize), each row has 255 columns, each column is a VARCHAR2(4000).

a) In prior releases (up to 11gR2), no matter the actual data content (e.g., fewer characters, NULL values), the JDBC driver allocates the maximum possible value for each column hence 8k bytes for VARCHAR(4000) (2 byte per character) times 255 columns times 1000 rows = 2040K bytes (or 2MB) per row x 1000 = 2GB char[].
b) With Oracle database 12c, the JDBC driver performs lazy allocation, in other words, it allocates only enough memory to store the metadata (15 bytes per value) + the actual data. If the value of a column is less than 4000, the driver will allocate 15 bytes of metadata + enough bytes to hold the actual value; if the value of a column is NULL then only 15 bytes will be allocated for that column; resulting in potentially significant memory saving when column data are less than the maximum possible value.

For a complete discussion, see the white paper "Oracle JDBC Memory Management" on the JDBC portal of Oracle Technology Network @ http://www.oracle.com/technetwork/database/application-development/index-099369.html

Large Network Buffers (SDU)

Session Data Unit (SDU) defines the size of internal buffers used by Oracle Net Services to move data from tables to database clients. If you can imagine moving GB of data using a 16KB buffer (the default), you will immediately understand the impact of SDU on query performance for Java applications retrieving large result sets of XML documents.
Oracle database 12c now supports very large network buffers up to 2 MB, up from 64K in Oracle database 11*g* Release 2. SDU can be configured at application level in JDBC URL, at database service level in TNSNAMES.ORA or globally at database level in SQLNET.ORA.
Note: SDU is not used when data is streamed directly to the client, without intermediate materialization e.g., SecureFile LOBs.

Database Resident Connection Pool

Database Resident Connection Pool (DRCP) is an RDBMS-side pool of servers processes shared across client applications, programming languages and middle-tiers. Introduced in Oracle database 11*g* OCI for C, C++, PHP, Python and Perl, it is indispensable for single-threaded systems (such as PHP/Apache) that cannot do client/middle-tier connection pooling. DRCP is also beneficial for multi-threaded systems such as Java, for large scale deployment of thousands of middle-tier accessing the same database. The reduction of database server processes and memory may reach orders of magnitude.

DRCP pools must be explicitly created configured, started and stopped by the DBA using the `DBMS_CONNECTION_POOL` package (available/installed out-of-the-box).

```
sqlplus /nolog
connect / as sysdba
execute dbms_connection_pool.start_pool();
execute DBMS_CONNECTION_POOL.CONFIGURE_POOL (session_cached_cursors=>50);
...
execute dbms_connection_pool.stop_pool();
```

DRCP is now available for Java through Oracle JDBC drivers. A client-side connection pool e.g., Oracle's Universal Connection Pool for Java is required for tracking connections check-in and check-out on the clients/mid-tiers.
New JDBC connection properties `oracle.jdbc.DRCP.name` and `oracle.jdbc.DRCP.purity`

allow Java applications to name DRCP pools and sub-partition a single pool across several applications.

To enable DRCP with JDBC and UCP, perform the following steps:

- Pass a non-null and non-empty string value to the connection property `oracle.jdbc.DRCPConnectionClass`
- Add `POOLED` qualifier to the URL in the short connection string
  Example: `jdbc:oracle:thin:@//localhost:5221/orcl:POOLED`
- Add (`SERVER=POOLED`) in the long connection string or in `TNSNAMES.ORA`

```
LongURL =
"jdbc:oracle:"+getTS()+":@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)"+"(HOST="+
getHost()+")(PORT="+getPort()+"))(CONNECT_DATA=(SERVICE_NAME="+
getSvcName()+")(SERVER=POOLED)))";
```

To enable DRCP with third party client-side connections pools, you must use new public methods under `oracle.jdbc.pool.OraclePooledConnection`. See the Oracle JDBC guide for more details.

Global Data Services for Java

Prior to Oracle database 12*c*, the scope of services such as Runtime Connection Load Balancing, Fast Connection Failover, web Session Affinity, Transaction Affinity were limited to single physical databases with multiple instances (i.e., RAC). Global Data Services (GDS) is a new framework which extends those services to database deployed anywhere within a globally distributed configuration (i.e., across geographies). Target databases configurations include Real Application Clusters (RAC), Active Data Guard, as well as configurations based on GoldenGate or other replication technology. Universal Connection Pool (UCP) have been enhanced to furnish Fast Connection Failover, Runtime Load Balancing, Web Affinity, and Transaction Affinity, across geographies.

How to Enable GDS

1. Enable Fast Connection Failover (FCF)
2. Automatic ONS configuration – i.e., no need to call `setONSConfiguration()`
3. Specify global service name and region in connect URL

```
( DESCRIPTION=
   ( ADDRESS_LIST= (LOAD_BALANCE=ON) (FAILOVER=ON)
     (ADDRESS=(GDS_protocol_address_information))
     (ADDRESS=(GDS_protocol_address_information))
   )
 (CONNECT_DATA= (SERVICE_NAME=global_service_name)
                              (REGION=region_name))
 )
```

Runtime Load balancing Enhancements

With Oracle database 12*c*, the Universal Connection Pool (UCP) brings significant enhancements to Runtime Load Balancing (RLB) in the areas of performance and smooth rebalancing. UCP now enforces strict Web and Transaction Affinity[6].

6 – Transaction Guard and Application Continuity for Java

Ensuring that customers do not pay twice a flight ticket, book or taxes is a universal usability requirement but a very hard technical problem to solve; similarly, capturing and replaying in-flight work in the face of database outage, is also a technically challenging problem to solve.

When a database outage occurs, four problems confront applications: (1) hangs, (2) errors, (3) determining the outcome of in-flight work and (4) the resubmission of in-flight work.

In "*Application Failover with Oracle database 11g*"[7], we described how applications can deal with hangs through Fast Application Notification (FAN). Oracle database 12*c* pushes the envelope further with: Transaction Guard for a reliable outcome of in-flight work and Application Continuity for capturing and replaying in-flight transactions.

New Concepts

Recoverable Error: Oracle database 12*c* exposes a new error attribute `is_recoverable` that applications can use to determine if an error is recoverable or not without maintaining their own list of error codes (e.g., `ORA-1033, ORA-1034, ORA-xxx`). JDBC throws `SQLRecoverableException` if the error is recoverable.

Database Request: a unit of work submitted by the application, including SQL PL/SQL, local calls, and remote procedure calls; has typically one COMMIT but could has zero or more than one.
Logical Transaction ID (LTXID): for determining the outcome of the last COMMIT statement.

Mutable Functions: non-deterministic functions that can change their results each time they are called e.g., `SYSDATE, SYSTIMESTAMP, SEQUENCES, SYS_GUID.`

FAN - HA Events & Notification

RAC and Data Guard emit HA events such as NODE DOWN, INSTANCE UP/DOWN, SERVICE UP/DOWN, etc; upon emission, these events are sent/notified to subscribers (drivers, applications) using Oracle Notification Services (ONS). Oracle JDBC drivers and the Universal Connection Pool subscribe to all HA events types when Fast Connection Failover is enabled and act upon. Java applications and third party drivers and connections pools may subscribe directly to DOWN events, using use `SimpleFAN.jar` (UP events are not currently supported).

[6] Affinity is covered in"Java developers Perspective on Oracle Database 11g" white paper.

[7] http://www.oracle.com/technetwork/database/app-failover-oracle-database-11g-173323.pdf

Transaction Guard for Java

Problems to solve

Address the 3rd issues that is: make a reliable determination of the outcome of the in-flight work. Following a break in communication between Java applications and the RDBMS, the outcome of last COMMIT operation is often doubtful and leads to the resubmission of work already committed thereby committing the same transaction twice or several times. This problem is challenging because simply checking the outcome at a given time does not guarantee a reliable outcome, as the COMMIT statement may eventually complete after the check.
Transaction Guard is an API for implementing "*at most one COMMIT"* by determining the outcome of the last COMMIT operation, in a fast, reliable and scalable manner; thereby ensuring that the execution of each logical transaction is unique.

Typical usage

1) Upon database instance crash: (i) death of sessions belonging to that instance; (ii) Fast Application Notification immediately sends the event to subscribers; (iii) application gets an error quickly; (iv) the connection pool (UCP) removes orphan connections from the pool

2) server-side package and procedure to help determine the outcome of the last COMMIT

a) New DBMS_APP_CONT package
b) Here is a sketch of the RDBMS and application interaction

```
If "recoverable error"
then
 Get last LTXID from dead session or from your JDBC callback
 Obtain a new database session
 Call DBMS_APP_CONT.GET_LTXID_OUTCOME with last LTXID to obtain
                    COMMITTED  and USER_CALL_COMPLETED status
 If COMMITTED and USER_CALL_COMPLETED
       Then return result
 ELSEIF COMMITTED and NOT USER_CALL_COMPLETED
       Then return result with a warning
 ELSEIF NOT COMMITTED
      Cleanup and resubmit request
       Note: the RDBMS prevents the transaction from committing (RETENTION_TIMEOUT)
 END
```

And here is the definition of GET_LTXID_OUTCOME

```
CREATE OR REPLACE PROCEDURE get_ltxid_outcome(
                  client_ltxid        IN  RAW,
                  committed           OUT INT,
                  user_call_completed OUT INT)
AS
   committed_b            BOOLEAN;
   user_call_completed_b  BOOLEAN;
BEGIN
 dbms_app_cont.get_ltxid_outcome(client_ltxid, committed_b,
                                                 user_call_completed_b);
 IF committed_b=TRUE THEN committed := 1;
```

```
 ELSE
    committed := 0;
 END IF;
 IF user_call_completed_b=TRUE THEN
      user_call_completed := 1;
 ELSE
      user_call_completed := 0;
 END IF;
END;
/
```

Ensure that execute permission on the DBMS_APP_CONT package has been granted to the database users that will call GET_LTXID_OUTCOME:

```
GRANT EXECUTE ON DBMS_APP_CONT TO <user-name>;
```

3) Application Usage (Java)

```
addLogicalTransactionIdEventListener()//register a listener to
                                  // Logical Transaction Id events

LogicalTransactionId firstLtxid = oconn.getLogicalTransactionId();
     //sent by the server in a piggy back message and hence this
     //method call doesn't make a roundtrip.
  …
CallableStatement cstmt = oconn.prepareCall(GET_LTXID_OUTCOME);
     // procedure defined above
  …
  committed = cstmt.getBoolean(1);
```

Supported Transaction Types

Transaction Guard supports the following transaction types: local transactions, DDL and DCL transactions, distributed and Remote transactions, parallel transactions, commit on success (auto-commit), and PL/SQL with embedded COMMIT.

Exclusions

In this release, Transaction Guard excludes the following transaction types

- Intentionally recursive transactions and autonomous transactions intentionally so that these can be re-executed.
- XA transactions
- Active Data Guard with read/write DB Links for forwarding transactions
- Golden Gate and Logical Standby

Configuration

RDBMS
On Service

- `COMMIT_OUTCOME`: values {`TRUE or FALSE`}, default is `FALSE`; applies to new sessions
- `RETENTION_TIMEOUT`: Units in seconds, default is 86400 (24 hours); maximum value is 2592000 (30 days)

```
SQL>
declare
params dbms_service.svc_parameter_array;
begin
  params('COMMIT_OUTCOME'):='true';
  params('RETENTION_TIMEOUT'):=604800;
  dbms_service.modify_service('[your service]',params);
end;
/
```

For a deeper coverage of Transaction Guard, please consult the “Transaction Guard” white paper on the Oracle Technology Network (OTN).

Application Continuity for Java

Problem to Solve

Address the 4th issue that confronts applications upon RDBMS instance failure, in other words, the resubmission of in-flight work resulting in masking database instance outage (hardware, software, network, and storage) to applications.

Solution

Application Continuity is a packaged solution which building blocks include: the unit of work (a.k.a. “*database request*”), the JDBC replay data source, Transaction Guard for Java, and RDBMS High Availability configurations (RAC, Data Guard).
Application Continuity works as follows:

1. Transparently captures in flight work a.k.a. “database request”, during normal runtime
2. Upon RDBMS instance outage or site failure, if recoverable errors then Transaction Guardis used under the covers then the driver reconnects to a good RDBMs instance (RAC) or disaster recovery `site (ADG)`
3. The driver and RDBMS cooperate to replay the in-flight work captured during normal runtime (until the point of failure).

When successful[8], Application Continuity masks hardware, software, network, and storage outages to applications; end-users will only observe/experience a slight delay in response time.

Table 1 below summarizes how Application Continuity works

[8] See restrictions and application design considerations in “Maximum Application Availability” or “Application Continuity” white papers

TABLE 1. PROCESSING PHASES OF APPLICATION CONTINUITY

NORMAL RUNTIME	RECONNECT	REPLAY
• Identifies database requests • Decides what is replayable and what is not • Builds proxy objects • Holds original calls with validation	• Ensures request has replay enabled • Handles timeouts • Creates a new connection • Validates target database • Uses Transaction Guard to enforce last outcome	• Replays held calls • During replay, ensures that user visible results match original • Continues the request if replay is successful • Throws the original exception if replay is unsuccessful

Configuration

In Oracle database 12*c* Release 1, *Application Continuity* is available with JDBC-Thin and UCP. These Oracle clients transparently demarcate units of work on connection check-out/check-in whereas third party drivers and connection pools must explicitly demarcate the units of work (a.k.a. "*database requests*") using `beginRequest()/endRequest()` calls.

Application Continuity for Java requires standard JDBC interfaces instead of deprecated `oracle.sql.*` concrete classes: `BLOB, CLOB, BFILE, OPAQUE, ARRAY, STRUCT,` or `ORADATA` (see My Oracle Support Note 1364193.1[9] for the deprecation notice).

1- Java Application
sets the new replay data source either in property file, as follows, or inline.
`datasource=oracle.jdbc.replay.OracleDataSourceImpl`

2- Enable Application Continuity on Service

```
FAILOVER_TYPE = TRANSACTION
REPLAY_INITIATION_TIMEOUT = 1800
FAILOVER_DELAY = 3 seconds
FAILOVER_RETRIES = 60 retries
SESSION_STATE_CONSISTENCY = DYNAMIC
COMMIT_OUTCOME = TRUE
```

Exclusions, Restrictions, and Design Considerations

Application Continuity exclusions, restrictions and design considerations are discussed further in Oracle "*Application Continuity*" white papers on the Oracle Technology Network (OTN).

[9] See JDBC interfaces for Oracle types:
https://support.oracle.com/CSP/main/article?cmd=show&type=NOT&id=1364193.1

Table 2 below summarizes restrictions

TABLE 2. WHEN IS APPLICATION CONTINUITY DEACTIVATED (NOT SUPPORTED)

GLOBAL	REQUEST	TARGET DATABASE
Any calls in same request after – • successful commit in dynamic mode (the default) • a restricted call • disableReplay API	• Error is not recoverable • Timeouts — Replay initiation timeout — Max connection retries — Max retries per incident • Target database is not valid for replay • Last call committed in dynamic mode	• Validation detects different results

Global Data Services - Connection HA Services

With GDS (described above), UCP has also been enhanced to furnish Fast Connection Failover across geographies. See the Oracle database 12c Active Data Guard white paper for more details on Global Data Services.

7 – Manageability, Ease of Use

Oracle database 12c furnishes row count per iteration for array DML, monitoring and tracing database operations, intelligent client connectivity and faster dead connection detection.

Row Count Per Iteration For Array DML

Java applications using Oracle JDBC drivers can now retrieve the number of rows affected by each iteration of an array DML statement (i.e., array INSERT, UPDATE, DELETE).

The following statement now prints the update count for each `UPDATE`.

```
…
int rcount[] = stmt.executeBatch();
```

Monitoring and Tracing Database Operations

For end-to-end tracing Oracle database furnishes a reserved namespace (OCSID) for storing tags: `MODULE, ACTION, CLIENTID, ExecutionContextID (ECID), MODULE, SEQUENCE_NUMBER` and the new `DBOP`.These tags may be associated with a thread without requiring an active connection to the database or client/server. When the application makes a database call, the tags are sent along to the database, piggybacking on the application's connection.

Java applications can use DBOP and other OCSID through either JDBC `setClientInfo()` method or DMS APIs
For example, you can set the value of the DBOP tag to foo in the following way:

```
...
```

```
Connection conn = DriverManager.getConnection(myUrl, myUsername,
myPassword);

conn.setClientInfo("E2E_CONTEXT.DBOP", "foo");
Statement stmt = conn.createStatement();
stmt.execute("select 1 from dual");  // DBOP tag is set after this
...
```

Intelligent Connectivity and Faster Dead Client Detection

Oracle Net Services is now smarter during connection attempts and decrease the priority of unresponsive nodes in the address string of connect descriptor, for subsequent attempts thereby increasing connectivity time and availability.

Similarly, the detection of terminated session/connection has been accelerated; the `SQLNET.EXPIRE_TIME` parameter in the `sqlnet.ora` configuration file helps detect terminated clients, faster. If the system supports `TCP keepalive`, then Oracle Net Services automatically uses the enhanced detection model, and tunes the `TCP keepalive` parameters.

8 - Security

Advanced Security Option

With Oracle database 12c, JDBC now supports SHA-2 hashing algorithms (including: SHA-256, SHA-384, and SHA-512) to generate secure message digests. Overall, Java applications can use the following hashing algorithms: MD5, SHA1, SHA-256, SHA-384 or SHA-512.

Usage

```
prop.setProperty
(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES,
"( MD5, SHA1, SHA256, SHA384 or SHA512 )");
prop.setProperty
(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL, "REQUIRED");
```

Customizing Default Java Security for Java in the database

The Java VM in Oracle database 12c enhances permission and policy management through a default policy configuration. In this release, the Java policy may be customized with adding third-party encryption suites then reloaded by the system administrator, as follows:

a) create $ORACLE_HOME/javavm/lib/security/java.security.alt
b) copy the contents of $ORACLE_HOME/javavm/lib/security/java.security into the newly created file
c) edit and load $ORACLE_HOME/javavm/lib/security/java.security.alt

```
cd $ORACLE_HOME/javavm
loadjava -u sys/<sys_pwd> -v -g public lib/security/java.security.alt
```

Runtime.exec with Java in the Database

For security reasons, it is advisable to run processes forked by `Runtime.exec` using OS identity with lesser rights. The following procedure associates a database user *DBUSER* with an OS *osuser* account:

```
dbms_java.set_runtime_exec_credentials('DBUSER', 'osuser', 'ospass');
```

9 - Applications Migration

Problem to solve: migrating Java applications built against foreign RDBMS. Oracle database 12*c* simplifies and reduces migration cost through new SQL types; however, the harder part is to support SQL syntax foreign to the Oracle database SQL engine.

Solution: a framework for translating foreign SQL syntax into Oracle SQL syntax before being submitted to the Oracle RDBMS SQL engine for compilation and execution.

SQL Statement Translation

The translation of foreign SQL statement syntax is performed by the *SQL Translation Framework* which consists in:
(i) a general purpose translation engine which runs in the Oracle RDBMS
(ii) a foreign RDBMS specific profile, which is plugged into the translation engine to allow translating the specific SQL dialect.

Java applications using Oracle JDBC drivers may now use the new SQL Translation mechanism, using the following new APIs:

- `oracle.jdbc.sqlTranslationProfile`
- `oracle.jdbc.sqlErrorTranslationFile`
- `oracle.jdbc.OracleTranslatingConnection`

And `SQLErrorTranslation.xml` configuration file.

Conclusion

This paper walked you through new Oracle database12c features for Java performance, scalability, availability and security; key features include support for the latest Java standards, embedded JVM support for multiple Java SE, Multitenant DataSource for Java, JDBC support for DRCP, large network buffers, support for new SQL types, Transaction Guard for Java, Application Continuity for Java, support for Global Data Service, advanced security features, finally, deprecated and de-supported features.

Java Performance, Scalability, Availability & Security with Oracle Database 12c
June 2013
Author: Kuassi Mensah
#kmensah

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com

Oracle is committed to developing practices and products that help protect the environment

Hardware and Software, Engineered to Work Together