What's New for Java DB, JDBC, and Database Web Services in Oracle Database 10*g*

An Oracle White Paper
May 2005

ORACLE

What's New for Java DB, JDBC, and Database Web Services in Oracle Database 10*g*

What's New for Java DB, JDBC, and Database Web Services in Oracle Database 10*g*

INTRODUCTION

Oracle Database 10*g* Release 1 brought major enhancements to the Java runtime in the database (OracleJVM), to the Oracle JDBC, and to the Oracle Database Web Services. Release 2 goes further and brings completeness in terms of standard support, integration with the Oracle database, manageability, and performance. In this paper I will give you an overview of Java, JDBC and Web Services features across Oracle Database 10*g*, (Release 1 and Release 2). Through these capabilities, database developers will: (i) reduce their costs through productivity gain; (ii) reduce their risks through increase scalability, load balancing, high-availability; and (iii) extend the outreach and reach-in of their Oracle databases. The white paper " *Java and Web Services Devlpers Perspective on Oracle Database 10g*" looks at those features from the developer's perspective.

WHAT'S NEW FOR JDBC

Reminder - JDBC Features in Oracle Database 10*g* R1

Standard Support

The Oracle9i Release 2 JDBC drivers initiates JDBC 3.0 support with the following features: *transaction savepoints, toggling between local and global transaction, reuse of PreparedStatement, and JDK 1.4.x support for client JDBC drivers.* The Oracle Database 10*g* Release 1 JDBC brought the following JDBC 3.0 features:
Named Parameters: named parameters in CallableStatement and PreparedStatement, will enable JDBC applications to pass parameters by name as well as registering, and retrieving output parameters by name.
New Ref interface and Datalink: a DATALINK value references a file outside of the underlying data source that the data source manages. The JDBC 3.0 specification maps this new JDBC data type to the Java type java.net.URL.
J2EE Connector Architecture Resource Adapter: the Oracle JDBC driver can function as a JCA resource adapter for Oracle databases.
Connection Pool: the Implicit Connection Cache (see below) is Oracle's implementation of JDBC 3.0 connection pool. Furthermore this implementation brought advanced capabilities such as connection tagging for faster search and retrieval, weight and attribute-based search

JDBC Web RowSet (JSR-114): this standard API allows applications such as Web Services clients or J2EE components to fetch a collection of rows from database tables (or other data sources) and emit the result set in XML format, disconnected from the Data-Source. The Oracle Database 10g R1 JDBC implementation is based on the draft specification for JSR-114.

Manageability / Scalability/ High-Availability/ Ease of Use / Performance

End-to-end Tracing: JDBC drivers to propagate the middle-tier Web client id to the RDBMS engine so as to trace resource consumption from a Web Browser to the SQL corresponding operations.

Enhanced Oracle JDBC Datum support: the separation of datum classes from the rest of jdbc for "small" download size for datum users. Better (finer-grain) SQLException handling; enhanced conversion and arithmetic operations methods on datum to-and-from other datum.

Instant Client JDBC-OCI drive: a downloadable, low footprint bundling for simplifying JDBC-OCI install

New database types: IEEE DOUBLE, and IEEE FLOAT: Java and J2EE applications will retrieve and perform faster arithmetic calculations without loss of information, using reduced database storage space.

LONG-to-LOB conversion: conversion function for CLOBs and BLOB to/from LONG, RAW and LONG RAW, will simplify their manipulation and improve their performance.

Unlimited size LOB: ever wanted to manipulate more than 4 Giga binary data?

Enhanced support for VARRAY: data mining applications will perform aggregation, collection and general set operations, more efficiently.

PL/SQL Index table: this feature lets you send and receive PL/SQL tables in the type-4 "thin" JDBC driver.

New encryption algorithms: the type-4 "thin" JDBC driver now supports `3DES112` and `3DES168`.

Direct XA: performance optimization of JDBC XA operations when using the type-4 "thin" JDBC driver.

JDBC-Thin Proxy authentication: multiple middle-tier users/threads can share the same database connection under the same authenticated user.

New JDBC Features in Oracle Database 10*g* R2

Standard Support

Oracle Database 10g JDBC Release 2 achieves 100% JDBC 3.0 compatibility by adding support for Auto-Generated Key Retrieval, Result Set Holdability as described below.

JDBC 3.0 Retrieval of Auto-Generated Keys: Oracle database sequences and triggers can be used to generate a unique key when a new row is inserted. JDBC 3.0 introduce three interfaces that support the retrieval of the auto-generated keys; namely

`java.sql.DatabaseMetaData, java.sql.Statement` and `java.sql.Connection`

- JDBC also offers "DML in the RETURNING clause" ss Oracle extension to support Retrieval of Auto Generated Keys.

JDBC 3.0 Result Set Holdability: this feature enables applications to decide whether the ResultSet object should be open or closed, when a commit operation is performed. The commit operation could be either implicit or explicit. The Oracle JDBC supports `ResultSet.HOLD.CURSORS_OVER_COMMIT` which is the default option for the Oracle database itself.

In addition, Oracle Database 10g JDBC Release 2 supports the final specification of *JDBC RowSet JSR-114,* a JDK 5.0 requirement. All 5 types of RowSets as JavaBeans: JdbcRowSet, CachedRowSet, WebRowSet, FilteredRowSet, and JoinRowSet are supported. In addition JAXP 1.2 (SAX 2.0 and DOM), and W3C XML schema, are also supported.

Manageability / Scalability/ High-Availability/ Ease of Use / Performance

JDBC-OCI support for Proxy Authentication: parity with JDBC-Thin Proxy Authentication (see above

JDBC-Thin support for SSL: JDBC leverages JSSE to provide a secure communication between client and server when specified (presence of (PROTOCOL=tcps) in the connection description).

JDBC-Thin support for TNSNAMES.ORA Lookup: similar to JDBC-OCI, both client-side and server-side JDBC-Thin can now read the local TNSNAMES.ORA file to resolve service name or TNS alias. Consequently Java in the database will also be in position to reference external Oracle database, using service name. See the JDBC documentation for more details.

Caching XAConnections: The JDBC connection cache has been enhanced to cache XAConnections, since this is a first implementation of such capability and we are looking for feedback (mostly Application Servers and ERP vendors).

New JDBC Connection Services

The Oracle Database 10g JDBC drivers introduce new connection services in term of performance, scalability and high-availability. In this section I'll describe the new Implicit Connection Cache (performance, scapalability), the Fast Connection Failover, (scalability, high-availability), and Runtime Load Balancing of JDBC Connection Requests (performance, scalability) in RAC and GRID environments,

Connection Cache Manager

The Connection Cache Manager provides a centralized way to manage one or more connection caches. It consists in a single instance of Connection Cache Manager per VM. Each Connection Cache Manager instance manages all connection caches

existing in the JDK/JVM. This instance of the Connection Cache Manager must be obtained before using any of the Connection Cache Manager functionality.

The Connection Cache Manager plays two major roles: cache management and binding connection caches to data sources.

Cache Management and Maintenance

The Connection Cache Manager is responsible for creating, destroying and maintaining the sanity of each cache. A programmatic API allows interaction with the Connection Cache Manager (i.e., explicitly create a cache). A connection cache may also be created transparently during the DataSource access. In fact, the first connection request on a DataSource attempts to create the cache, however, if it already exists then, the cache creation is skipped and the connection request is simply routed to the cache, which retrieves a connection.

Binding a Connection Cache to the DataSource

The Connection Cache Manager associates a connection cache with its DataSource, during the creation and re-initialization of the cache. The associations ends when the cache is removed/destroyed. This ensures an efficient access and connections retrieval when `getConnection()` request is invoked on the DataSource object.

Multi-cache Support: The Connection Cache Manager supports coexistence of more than one cache. Each cache identified by a unique name, is tightly bound to a DataSource. Each cache is either created transparently when getConnection() requests are made on a cache enabled DataSource or is created explicitly in the middle tier via the Connection Cache Manager API. Using multiple cache enabled DataSources, provides the following benefits:

Request connections to more than one DataSource, especially when each of these DataSources point to a separate underlying database.

Manage all caches configured, via the Connection Cache Manager. This makes managing caches easy, without the burden of unnecessary book keeping in Application Server code (code using Connection Cache Manager).

There are no limits imposed by the Connection Cache Manager on the number of caches that can be created. This number is limited by the resources available on the JVM and the Oracle database server.

Once a cache is created, it may either be explicitly removed via the Connection Cache Manager, or is removed when the DataSource is closed after use, via the DataSource close() API.

Connection Cache Properties

It is easy to set limits and tuned the connection cache using a set of connection cache properties. Cache properties are set either while creating the cache for the first time, on the DataSource, or when the re-initializing the cache, via the Connection Cache Manager.

MinLimit: this sets the minimum number of PooledConnections the cache maintains. This guarantees that the cache will not shrink below this minimum limit. This property does not initialize the cache with minimum number of connections[1]. Refer to ***InitialLimit*** property for priming of cache information. The default value is 0.

MaxLimit: this sets the maximum number of ***PooledConnections*** the cache can hold. The default value is unbound, meaning that there is no MaxLimit assumed. Since the connection cache does not assume the maximum limit, the connection cache is only limited by the number of database sessions configured for the database. In other words, connections in the cache could reach as much as the database allows.

InitialLimit: this sets the size of the connection cache when the cache is initially created or reinitialized. When this property is set to a value greater than 0, that many connections are pre-created and ready to use. This property is typically used to reduce the "ramp up" time in priming the cache to its optimal size. The default value is 0.

Dynamic Reconfiguration Support

Implicit Connection Cache allows dynamic reconfiguration of the cache properties by reinitializing the cache using the specified new set of cache properties. The new properties take effect on all ***PooledConnections*** newly created as well as *PooledConnections* that are not in use. For connections that are in use, the new properties take effect only after they are returned to the cache.

MONITORING THE CONNECTION CACHE

These APIs may be invoked on the Connection Cache Manager to obtain cache information, such as the number of connections checked out, or the number of connections available in the cache.

int getNumberOfAvailableConnections(String cacheName)

This API returns the number of connections in the connection cache, that are available for use. The value returned is a snapshot of the number connections available in the connection cache at the time the API was processed and hence a statistical value.

int getNumberOfActiveConnections(String cacheName)

This API returns the number of checked out connections. These are connections that are active or busy, and hence not available for use. The value returned is a snapshot of the number of checked out connections in the connection cache at the time the API was processed and hence a statistical value.

java.util.properties getCacheProperties(String cacheName)

[1] As was the case with OracleConnectionCacheImpl

Retrieves the properties of the specified cache.

`String[] getCacheNameList()`

This API returns all the connection cache names that are known to the Connection Cache Manager. The cache name(s) may then be used to manage connection caches using the Connection Cache Manager APIs.

Implicit Connection Cache

The Implicit Connection Cache provides a rich set of features including: transparent or implicit access to the connection cache, support for heterogeneous authenticated connection, the ability to refresh or recycle stale connections from the cache, connection retrieval based on user defined attributes, connection retrieval based on attributes and weights.

Abandoned Connection Support

Abandoned or orphaned connections are those that have been checked out of the connection cache, but never returned to the cache, for numerous reasons. Implicit Connection Cache automatically detects and reclaims these connections back to the cache, based on the `AbandonedConnectionTimeout` property on the connection cache. Setting this property to a valid timeout value, starts a heartbeat monitoring process[2] on this connection that is checked out.. When a heartbeat is not registered on the connection for the specified period of time, the connection is considered abandoned and is automatically claimed and put back to the cache.

Statement Caching Support

The `MaxStatementsLimit` connection cache property sets the maximum statements to keep open for a connection cache. If cache is reinitialized with this property and if more cursors are cached than specified, the extra cursors are closed. The default is 0.

Connection Recycling Support

Over a period of time, connection cache accumulates stale connections. There are two modes for recycling or refreshing stale connections in the cache: `REFRESH_INVALID_CONNECTIONS` and `REFRESH_ALL_CONNECTIONS`.

- With `REFRESH_INVALID_CONNECTIONS` each PooledConnection in the cache is checked[3]; if invalid, the connection's resources are removed and replaced with a new PooledConnection.
- WIth `REFRESH_ALL_CONNECTIONS`, all the available connections in the cache are closed and replaced with new valid physical connections.

Connection Retrieval based on User defined Attributes

---

[2] Any SQL activity on the connection
[3] The validity test is as simple as: *select 1 from dual;*

The implicit connection cache supports the idea of striping (i.e., tagging) a connection with user-defined attributes before returning it to the cache. These attributes can later be used to retrieve the same connection from the cache.

Example 1: Retrieval based on Connection Attribute NLS_LANG

```
// Look up the datasource object
javax.sql.DataSource ds = (javax.sql.DataSource)
ctx.lookup(MyOracleDataSource);

// get a connection from MyCache
java.util.Properties connAttr = null;
connAttr.setProperty("NLS_LANG", "ISO-LATIN-1");
conn = ds.getConnection(connAttr); // retrieve
connection - NLS_LANG
…
conn.applyConnectionAttributes(connAttr); // apply
attributes

Statement stmt = conn.createStatement();
stmt.execute("select empname from emp");
…
conn.close(); // release the connection back to MyCache
```

Example 2: Retrieval based on Connection Attribute Isolation level

```
…
java.util.Properties connAttr = null;
connAttr.setProperty("TRANSACTION_ISOLATION",
"SERIALIZABLE");
conn = ds.getConnection(connAttr); // retrieve
connection that matches Transaction Isolation
…
conn.close(connAttr); // another way to  apply
attributes to the connection
```

Example 3: Retrieval based on Connection Attribute, Connection tags (Reserved connection)

```
…
java.util.Properties connAttr = null;
connAttr.setProperty("CONNECTION_TAG",
"JOE'S_CONNECTION");
conn = ds.getConnection(connAttr); // retrieve
connection that matches Joe's connection
…
conn.close(connAttr); // apply attributes to the
connection
…
conn = ds.getConnection(connAttr); // This will retrieve Joe's connection
```

Connection Retrieval Based on Attributes and Weights

The cache property `CacheAttributeWeights` are `java.util.Properties` that allows setting attribute weights. Weights are assigned to each Key in a `ConnectionAttribute`. Each weight is an integer value that defines the cost of the Key. Once the weights are specified on the cache, connections may be retrieved using

`getConnection(connectionAttributes)` which searches for a connection that satisfies a combination of:

- Key/Value match on a connection from the cache
- Maximum total weight of all the keys of the `connectionAttributes` that were matched on the connection

Consider the following example.

A cache is configured with `CacheAttributeWeights` as follows:

```
java.util.properties cacheProps = new Properties();
java.util.properties cacheWeights = null;

cacheWeights.setProperty("NLSLANG", "10");
cacheWeights.setProperty("SecurityGroup", "8");
cacheWeights.setProperty("Application", "4");
…
// set weights on the cache
cacheProps.put(CacheAttributeWeights, cacheWeights);
                    …
```

Once the weights are set, a connection request could be made as:

```
java.util.properties connAttr = null;
connAttr.setProperty("NLSLANG", "ISO-LATIN-1");
connAttr.setProperty("SecurityGroup", "1");
connAttr.setProperty("Application", "HR")

// Request connection
ds.setCacheName("MyCache");
// First retrieval of connection from myCache
conn = ds.getConnection(connAttr);
…
conn.close(connAttr); // apply attributes on the
connection
…
// Next retrieval finds the connection in the cache
conn = ds.getConnection(connAttr);
…
```

This *`getConnection()`* request tries to retrieve a connection from the cache, *`MyCach`e*. The connection matching and retrieval from the cache involves the following:

- An exact match: a connection that satisfies the same attribute values and all the Keys (NLS_LANG, SecurityGroup, and Application).
- An exact match is not found: if the *`ClosestConnectionMatch`* property is set, a closest match based on the attribute key/value and their associated weights are used. For example, a closest match may be a connection that contains attribute match of NLS_LANG and APPLICATION, but not SECURITY_GROUP. It is also possible to find connections that match some keys of the original list, but their combined weights are the same. For example, connection1 could have a match of NLS_LANG with its associated weight of 10, where as connection2 may have an attribute match of SECURITY_GROUP and APPLICATION with their combined weight of 12. In this case, it is desired that connection2 is returned. In other words,

connection2 is the closest match and more expensive to reconstruct (from the caller's perspective) as opposed to connection1. When none of the *connectionAttributes* match a new connection is returned. The new connection is created using the user and password set on the DataSource.

Once the connection is returned, the user can invoke `getUnMatchedConnectionAttributes()` API on the connection object to return a set of attributes (`java.util.Properties`) that did not match the criteria. These unmatched attribute list is used by the caller (or application) to re-initialize these values before using the connection.

Applying Connection Attributes to a Cached Connection

There are two ways to apply a connection attribute to a connection. (1) By calling `applyConnectionAttributes(java.util.properties connAttr)` API on the connection object. This simply sets the supplied attributes on the connection object. Its possible to apply attributes incrementally using this API, allowing users to apply connection attributes over multiple calls. For example, *NLS_LANG* may be applied by calling this API from module A. The next call from module B, may then apply the *TXN_ISOLATION* attribute, and so on. (2) By calling `close(java.util.properties connAttr)` API on the connection object. This API closes the logical connection and then applies the supplied connection attributes on the underlying PooledConnection (physical connection). The attributes set via this `close()` API overrides the attributes, if any, set using the `applyConnectionAttributes()` API.

The following example shows a call to `close(connectionAttributes)` API on the connection object, that lets the cache apply the matched `connectionAttributes` back on the PooledConnection, before returning it to the cache. This ensures that when a subsequent connection request with the same connection attributes is made, the cache would find a match.

// Sample connection request and close

```
java.util.properties connAttr = null;
connAttr.setProperty("NLSLANG", "ISO-LATIN-1");
conn = ds.getConnection(connAttr); // request connection
based on attributes
java.util.properties unmatchedAttr =
conn.getUnMatchedConnectionAttributes();
…
--- App Server code applies unmatched attributes to the
connection, either by
    calling PL/SQL procedures or SQL, before using the
connection. ---
--- work ---
…
conn.close(connAttr); // apply attributes to connection
```

Fast Connection Fail-Over

Fast Connection Failover works in conjunction with the Implicit connection caching mechanism and a RAC database.

When a connection cache is setup for a multi-instance RAC database, database connections may end up on any of the RAC instances, as distributed by the database listener. When an instance goes down, due to an instance or host failure, it brings down all the connections with it. For example, in a two node, two instance RAC cluster, consider a scenario where each of the instances were to have 50 connections each, for a connection cache primed up to 100 connections in the cache. If one of RAC instances goes down, there would instantly be 50 bad connections in the cache. This could potentially result in multiple bad connections being returned from the cache, which in turn could result in application or browser page errors.

The Fast Connection Failover support enables an automatic detect-and-fix mechanism to handle any instance or host failures in a RAC environment. The mechanism is enabled on a cache enabled DataSource, by simply flipping the DataSource property *FastConnectionFailoverEnabled* to *true*.

```
// Example to show binding of OracleDataSource to JNDI
// with relevant cache properties set  on the
DataSource.

import oracle.jdbc.pool.*; // import the pool package

Context ctx = new IntialContext(ht);
OracleDataSource ods = new OracleDataSource();

// Set DataSource properties
ods.setUser(“Scott”);
ods.setPassword(“tiger”);
ods.setConnectionCachingEnabled(True);
ods.setConnectionCacheName(“MyCache”);
ods.setConnectionCacheProperties(cp);
setURL("jdbc:oracle:thin:@(DESCRIPTION=
   (LOAD_BALANCE=on)
   (ADDRESS=(PROTOCOL=TCP)(HOST=host1) (PORT=1521))
   (ADDRESS=(PROTOCOL=TCP)(HOST=host2)(PORT=1521))
   (CONNECT_DATA=(SERVICE_NAME=service_name)))");
ods.setFastConnectionFailoverEnabled(true); // Enable
fast connection failover
ctx.bind(“MyDS”, ods);
…
ds = lookup(“MyDS”);  // lookup DataSource from the
cache
// implicitly create connection cache, that is set up
for fast connection failover
conn = ds.getConnection();
…
conn.close();  // return connection to the cache
…
ods.close() // close datasource and cleanup the cache
```

The Fast Connection Failover mechanism works by handling Service DOWN or UP events and Host DOWN events. The DOWN event processing always cleans up the bad connections from the cache. The UP event processing does Load Balancing of connections in the cache.

The Fast Connection Failover functionality provides the following advantages:

Fast shutdown of connections in the connection cache, when RAC instance/node failures are detected: this prevents bad or invalid connections being handed out to application connection requests. For an application that depends on the Implicit connection cache for total connection management, the Fast Connection Failover mechanism provides maximum connection availability.

Load Balancing of connections when a RAC UP event is generated: in this model, connections are established and load balanced to all active RAC instances, without waiting for application connection retries/requests. Note that for RAC setups, the service is almost always instantly available, except when the entire service is down.

Runtime Load-Balancing of Connection Requests

In Oracle Database Release 1, connections to RAC instances were randomly established by the TNS Listener, without any consideration of how busy an instance already was. In Oracle Database 10g Release 2, RAC nodes emit Service Metrics every 30 seconds (i.e., RLB events), for each service on each instance. Each event payload contains a state or health of the service in question and the percentage of idleness (in other words, the percentage of additional work related to the service in question that the instance can handle). The RAC Automatic Workload Repository (AWR) contains the directives for the workload management goal (i.e., throughput or service time). The Oracle JDBC connection cache and the Connection Cache Manager have been extended to exploit RAC service metrics. When RLB is enabled (see configuring RLB, later), a connection belonging to the least loaded instance for the service is retrieved from the cache and returned to the requestor. By routing connections requests based on workload feedback, the workload will be evenly distributed across all nodes that furnish the service in question; thereby optimizing resource utilization (which is the whole purpose of Enterprise Grid Computing).

Configuring Runtime Load-Balancing

The connection pool Manager routes the connection request to the least loaded RAC instance using different policies.

WHAT'S NEW FOR JAVA-IN-THE-DATABASE

Why Java In the Database?

Java in the database is a response to customers in quest for portability, reuse and the ability to do things that cannot be done in PL/SQL (a lot of things!). Java is indeed portable across hardware platform, across RDBMS (no vendor lock-in), and across tiers – furthermore Java can be migrated to/from middle-tier (J2EE, POJOS, JDBC) to the database (Java stored procedures). Other important benefits Java brings are the ability to just reuse, the huge Java class libraries produced over years by millions of developers; and also the ability to reuse the pervasive Java skills.

Reminder – Java DB Features in Oracle Database 10*g* R1

On standard support front, the OracleJVM runtime achieved full compatibility with J2SE 1.4 including stringent Java Security (JCE, JSSE, etc), headless AWT, JAXP, etc. On the performance front, OracleJVM implements self-tuning Java pool; optimized memory management (end-of call migration) for dedicated processes; implemented a new, faster server-side JDBC driver; rewrote a new, faster bytecode verifier (loadjava). Fixed EJB Call-out, which completed the set of call-out capabilities: HttpClient, RMI/JRMP, RMI/IIOP, and SOAP Client. JPublisher utility now offers the generation of client-side stub for direct invocation of Java in the database, without a user-supplied Call Spec (PL/SQL Wrapper).

New Java DB Features in Oracle Database 10*g* R2

Optimize OracleJVM Garbage Collector behavior

A new parameter PGA_AGGREGATE_TARGET influences OracleJVM GC behavior at the end of the SQL call. Low values will optimize memory at the expense of the speed while high values will optimize execution speed at the expense of memory.

Java Audit

With pre-Release 2 of Oracle Database 10g, only Java classes/methods published to SQL through PL/SQL wrapper can be audited through SQL Audit. With this enhancement, java sources, java classes or java resources can be directly/explicitly audited. The following table summarizes the options and scope of Java audit.

Java Audit Option	SQL Statement to be Audited
CREATE JAVA SOURCE	CREATE JAVA SOURCE CREATE OR REPLACE JAVA SOURCE
ALTER JAVA RESOURCE	ALTER JAVA RESOURCE
DROP JAVA SOURCE	DROP JAVA SOURCE

CREATE JAVA CLASS	CREATE JAVA CLASS CREATE OR REPLACE JAVA CLASS
ALTER JAVA CLASS	ALTER JAVA CLASS
DROP JAVA CLASS	DROP JAVA CLASS
CREATE JAVA RESOURCE	CREATE JAVA RESOURCE CREATE OR REPLACE JAVA RESOURCE
ALTER JAVA RESOURCE	ALTER JAVA RESOURCE
DROP JAVA RESOURCE	DROP JAVA RESOURCE

WHAT'S NEW FOR JPUBLISHER & DATABASE WEB SERVICES

Why Database Web Services

Database Web Services bring three benefits to database users:

- Database as Web Services Provider extends database's client-base to Web Services clients by allowing the execution of database operations and data retrieval through Web Services mechanisms.
- Database as Web Services Consumer extends database's reach. The inclusion of external data as part of a SQL query or database batch processing is not new (see gateways and other related mechanisms), what's new is when the external data is only available as dynamic data (produced on demand), typically available as a Web services (i.e., temperature, stock price, IRS table, etc).
- Reusing the Web Services framework of Oracle Application Server with the database, allows instant interoperability, consistent Web services development and deployment; furthermore it integrates the Database in the Service Oriented Architecture beyond basic SOAP/HTTP; Database Web Services inherit Web Services Interoperability, WS-Security, WS-Reliability, and WS Management.

Reminder -- JPublisher and Database Web Services Features in Oracle Database 10*g* R1

JPublisher

Complete JDBC types support: new supported types including: NCHAR, Timestamp, SQLJ objects, and SQL Opaque. Easier access to native PL/SQL types through predefined type conversions. And the generation of client stub for invoking Java-in-the-database.

Database Web Services

Support for Database as Services Consumer

For a given WSDL, generate the Java proxy classes, the PL/SQL wrappers and load the appropriate files in the database.

Support for Database as Web Services Provider

- Publishing PL/SQL, Java Stored Procedure, SQL Queries, and SQL DML as Web Services
- Mapping PL/SQL Types (CLOB, BLOB)
- Mapping of REF CURSORs and Result Sets

New JPublisher and Database Web Services Features in Oracle Database 10*g* R2

- Support JDBC types for server-side Java calling
- Publishing Streams/Advance Queue as Web Services
- Support complex types in Web Services callout
- Handle connection loss through data sources
- Generate SQLJ runtime free code (works as pure JDBC code)

CONCLUSIONS

This paper gave you a brief overview of Java, JDBC and Database Web services enhancements in Oracle Database 10g release 1 and release 2. The net benefits are: faster application development, low cost application development, faster applications execution, low cost platform integration, dynamically scaling up and down RAC/Grid configurations, extending the reach of the database, extending the capabilities of the database, reuse of existing database entities as Web Services.

What's New for Java DB, JDBC, and Database Web Services in Oracle Database 10*g*
May 2005
Author: Kuassi Mensah

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