Author : Chris Drawater
Date : Oct 2007
Version : 1.0

Notes on using PostgreSQL 8.2.4 for J2EE/JDBC applications

Abstract

A basic overview of some of the changes required to port JDBC applications from Oracle to PostgreSQL.

Document Status

Introduction

This paper documents provides a basic overview of porting JDBC applications from Oracle to PostgreSQL. XML/XQuery is not covered.

These notes are an update on an earlier document (Ref 1) with new sections on JDBC3 returning keys, server prepared statements and error handling.

It is based upon experience with the following configurations :

Databases →
        Oracle 10.2
        PostgreSQL 8.1 & 8.2.4

Development Environment on Windows XP →
        PostgreSQL JDBC driver - *postgresql-8.1-404.jdbc3.jar*
        JDK 1.5.0
        Apache 2.0.55
        Tomcat 5.5.15
        Connector : Apache Tomcat JK 1.2.15 for WIN32 – works with Apache 2.0.55 and later
        Orion AS 2.0.2

On Solaris 10 (x86) →
        PostgreSQL JDBC driver - *postgresql-8.2-505.jdbc3.jar*
        JDK 1.5.0

For demonstrative purposes, 'vAuth' is used as the name of the application.

Abbreviation & Definitions

AS → Application Server *(for simplicity including Tomcat)*
PG → PostgreSQL
OLTP → Online Transaction Processing   (ie. no data trawling or MIS etc)
MPP → Massively Parallel Processing or Processor
VLDB → Very Large Database
MVCC → Multi-Version Concurrency Control
DDL → (SQL) Data Definition Language

Positioning

PostgreSQL 8.2 in it's standard form (as downloaded from http://www.postgresql.org/download) is arguably best suited to OLTP and small datamarts or reporting for small/medium data volumes of say arbitrarily up to 100's of Gb of data *(large being when manipulating/managing/backing up the data volume becomes problematic)*.

Currently, standard PostgreSQL 8.2 does not have the server level parallel operations or **inbuilt** MPP type capability nor some of the diagnostic information available that would allow it to move up into the VLDB data warehouse space.

Background for Oracle Developers

For developers coming from an Oracle background, PostgreSQL has a number of familiar (often near identical) concepts including
> MVCC
> The same transaction isolation levels with a default of "read committed"
> Optional table level locking ('*lock table…*')
> Default Row level locking for data writes
> Btree indexes (also other index types available)
> Referential integrity (primary, foreign keys)
> Triggers
> Sequence numbers
> Explain (for looking at problem queries etc) & optimizer statistics
> Views

also
> DBMS server side functions/procedures (available in a variety of languages)

Also available within PostgreSQL, but not quite the same as in Oracle and so needing a little more consideration, are
> Query rewrite (Oracle) & Rules (PostgreSQL)
> Types (PostgreSQL is far more extensive)
> Table inheritance
> Roles
> Java Stored Procedures (not in base product, but available following the links at PostgreSQL: Downloads)
> 2 phase commit support
> varrays (although not for composite data types)

Developers should not find the switch from Oracle to PostgreSQL too problematic for OLTP type systems.

However, be aware is that the following Oracle type technologies are not available with PostgreSQL 8.2.4 :

No bitmap indexes
No materialized views
No parallel options on DDL etc
No parallel query
No packages
No DB links
No distributed queries
No synonyms
No Index Organized Tables (IOT)

Command Line SQL Interface

The equivalent of the Oracle *sqlplus* utility is the PostgreSQL *psql* utility, which (assuming the environment has been set up correctly) can be invoked by

$ psql <DB> <User>

Note that, by default, auto-commit is enabled, so to execute a multi-statement TX, use either

begin work;
        SQL etc
commit;

or

\set AUTOCOMMIT OFF
        SQL etc
commit;

Note that auto-commit can be turned off either programmatically within JDBC code (see later) or sometimes within the AS specific DataSource definitions, so Java application code doesn't need to be modified.

Converting SQL DDL from Oracle to PostgreSQL

Many of the PostgreSQL datatypes will be familiar to Oracle and ANSI SQL developers.

As a starting point, approximate equivalent datatypes are as follows , but please check the documentation to verify datatype precision and exact meaning, and datatype comparison semantics etc.

ANSI	PostgreSQL 8.2	Oracle 10g
integer,	integer	number
numeric, decimal	numeric, decimal	number
float	float	number
char	char	char
varchar	varchar	varchar2
date	date	date (includes time to sec)
	timestamp	timestamp
	bytea	BLOB
	text	CLOB

Tablespaces can be specified   for table or index creation , but there are no Oracle type storage parameters : only the tablespace name   (which maps down to a filesystem directory) is required.

For example,

> *create index auth_expiry on UserAuthentication (expiry)*
> *tablespace APPDATA;*

The familiar Btree index is available, including partial , multi-col, and unique variants, as is   standard referential integrity (primary, foreign keys).

PostgreSQL partitioning is not as slick as that of Oracle – basically it relies upon   table inheritance with each sub-table (equivalent to a partition) having an optimizer aware contraint which defines the range or list of key values which in turn defines/controls the contained data.. Please see the PostgreSQL documentation for further information.

Whenever possible, use ANSI or common SQL datatypes and DDL.

JDBC driver

A pure Java (Type 4) JDBC driver implementation is available and can be downloaded from
http://jdbc.postgresql.org/

Assuming the use of the JDK 1.5, download
postgresql-8.1-404.jdbc3.jar or *postgresql-8.2-505.jdbc3.jar*

and make the driver available to the application server classpath.

For Orion 2.0.2, copy to *ORION_BASE/lib* .
For Tomcat 5.5.15, copy the file to *TOMCAT_HOME\common\lib*

(If moving JAR files between different hardware types, always *ftp* in *BIN* mode).

J2EE Application Servers – Configuring DataSources

Configuring a PostgreSQL DataSource is little different from any other database DataSource but is usually
AS vendor dependant.

Below is an example of a DataSource configuration for the Orion 2.0.2 AS and this XML definition
would be included in file *$ORION_BASE/config/data-sources.xml*.

```
        <data-source
                class="com.evermind.sql.DriverManagerDataSource"
                name="vAuthDS"
                location="jdbc/vAuthDS"                     <!-- JNDI path for  basic DataSource -->
                pooled-location="jdbc/vAuthPooledDS"       <!-- JNDI path for  pooled  DataSource -->
                xa-location="jdbc/xa/vAuthXADS" <!-- JNDI path for XA DataSource -->
                ejb-location="jdbc/vAuthEJBDS"             <!-- JNDI path for  EJB DataSource -->
                connection-driver="org.postgresql.Driver"
                username="xyz"
                password="xyz"
                url="jdbc:postgresql://10.248.42.78:5432/db9"
                max-connections="5"                         <!-- max pool size -->
                min-connections="3"                         <!--- min pool size  -->
                inactivity-timeout="300"                    <!-- 5 mins  -->
        />
```

The *DriverManagerDataSource class* is the wrapper class which allows Orion to use the PostgreSQL
implementation of a Connection driver as a DataSource.

With Tomcat 5.5.15, to configure an PostgreSQL DataSource specific to an application (ie not defined globally), create a *context.xml* file containing :

```
<Context>
        <Resource
                auth="Container"
                description="vAuth Postgresql DB Connection"
                name="jdbc/vAuthDS"
                type="javax.sql.DataSource"

                username="xyz"
                password="xyz"
                driverClassName="org.postgresql.Driver"
                url="jdbc:postgresql://10.248.42.122:5432/db9"

                initialSize="3"
                maxActive="10"
                maxIdle="5"
                minIdle="3"
                maxWait="5000"

                validationQuery=""
                poolPreparedStatements="false"
        />
</Context>
```

This application specific file *context.xml* (as per above) needs to be created under META-INF (alongside WEB-INF) in the WAR .

The hierarchical application WAR directory tree should look something like

```
<app root>
        <app root>/*.jsp                                    files
        <app root>/*.html                   files
        <app root>/*.gif                    files
        <app root>/*.jsp                                    files
        <app root>/WEB-INF                  dir
                <app root>/WEB-INF/web.xml          file
                <app root>/WEB-INF/classes          dir
                <app root>/WEB-INF/lib              dir
                        <app root>/WEB-INF/*.jar    files
        <app root>/META-INF                 dir
                <app root>/META-INF/context.xml     file
```

To enable the application to reference the Tomcat managed DataSource, a resource XML entry (matching the DataSource defined in *context.xml*) must be placed in the application *web.xml* file – for example :

```
<resource-ref>
        <description>vAuth Datasource</description>
        <res-ref-name>jdbc/vAuthDS</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
</resource-ref>
```

Using JDBC DataSources

A JDBC DataSource is usually accessed via a JNDI lookup.

Again the JNDI path may be AS vendor implementation specific, but other than that, the basic code should not change.

A very simple example of application code acquiring a pooled database *Connection* object via a *DataSource* using a JNDI lookup would look something like :

```
        String dsString = "java:/comp/env/jdbc/vAuthDS";          // Tomcat

        Context ic = new InitialContext();
        DataSource ds = (DataSource) ic.lookup(dsString);

        Connection con = ds.getConnection();
```

Direct JDBC Connections

If non-DataSource derived Connection objects are used, then the URL used to connect to the PostgreSQL server should be of the form

        jdbc:postgresql://host:port/database

As seen in an earlier section, this URL should also be used within DataSource definitions.

Replace the line (used to load the JDBC driver)
        Class.forName ("oracle.jdbc.driver.OracleDriver");
with
        Class.forName("org.postgresql.Driver");

and remove any Oracle specific imports, such as
        import oracle.jdbc.driver.*;

JDBC Connection Setup

Not really PostgreSQL specific issues , but at the *Connection* level , it is also advisable to switch off the *autocommit* feature

> *Connection con;*
> *...*
> *con.setAutoCommit(false);*

and set the default isolation level to "read committed"

> *con.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);*

This setup provides a default TX behavior that mirrors that of Oracle.

JDBC Extensions

Remove any Oracle JDBC extensions, such as
> *((OracleConnection)con2).setDefaultRowPrefetch(50);*

Instead, the row pre-fetch must be specified at an individual *Statement* level →

eg.   *PreparedStatement pi = con1.prepareStatement("select....");*
> *pi.setFetchSize(50);*

If not set, the default fetch size will default to 0;

For , PostgreSQL to make use of server prepared statements after a specified number of executions, then on a *Connection* wide basis, use

> *Connection con;*
> *((PGConnection)con).setPrepareThreshold(1);*   // use prepared statements upon 1st
execution

This facility may also be enabled at an individual *preparedStatement* level – see the JDBC driver documentation.

JDBC3 – returning automatically generated keys

JDBC3 defines the facility to return a database generated key from an SQL insert. This has been implemented in the Oracle JDBC driver but (at the time of writing) not yet in the PostgreSQL driver.

The workaround is to perform the SQL *insert* immediately followed by a SQL *select* which , using the same column values to uniquely identify the single row , returns the generated unique key. An underlying unique multi-column index referencing these columns may also be required.

So any Oracle SQL *insert* , returning a key generated by a database sequence, for example similar in nature to →

```
Connection con;
long ecode;

String rvs[] = {"ecode"};  // only 1 value to be returned == named column
PreparedStatement ws = con.prepareStatement("insert into equipment(ecode,model,…)
        values (sid.nextval,?,…)",rvs);

ws.setString(1,model);
…
ws.executeUpdate();
ResultSet rs = ws.getGeneratedKeys();
while (rs.next())
{
        ecode = Long.parseLong(rs.getString(1));    //getLong(1) fails with error
}
rs.close();
con.commit();
```

needs to be modified with the addition of SQL *select* , for example →

```
PreparedStatement ss = con.prepareStatement("select ecode from equipment
                        where model = ? …");
PreparedStatement   ws = con.prepareStatement("insert into equipment(ecode,model,…)
        values (nextval('sid'),?,…)");

ws.setString(1,model);
…
ws.executeUpdate();
con.commit();

ss.setString(1,model);
…
rs = ss.executeQuery();
while (rs.next())
{
        ecode = rs.getLong(1);
}
rs.close();
```

Also note the difference in the way database sequences are referenced (see later section).

SQLState & Error Handling

The SQLState codes for Oracle and PostgreSQL may not necessarily be exactly the same.

For example, a unique key violation
Oracle → 23000
PostgreSQL → 23505

So check any error handling code, such as below, for reliance upon hard coded SQLState values.

```
try
{
        ...
        ws.executeUpdate();
        con.commit();
}
catch (SQLException ex)
{
        if (ex.getSQLState().equals("23505"))          // PG duplicate key
        {
                con.rollback();
        }
        else // propogate upwards
        {
                throw new SQLException("..."+ ex.getMessage() );
        }
}
```

See the PostgreSQL 8.2 documentation for a list of the SQLState codes.
Test your code thorough for all possible error scenarios…!

Oracle's SYSDATE in SQL DML

Sysdate can be replaced with '*now'::timestamp*.

For example,

insert into UserAuthentication(...,expiry) values (..., sysdate + 10);

can be replaced by
insert into UserAuthentication(...,expiry) values (..., 'now'::timestamp + '10 day');

Oracle SQL Extensions

Any non ANSI SQL extensions will need changing.

For example sequence numbers
        Oracle => *online_id.nextval*
should be replaced by
        PostgreSQL => *nextval('online_id')*

Oracle 'hints' embedded within SQL statements are ignored by PostgreSQL.

Wherever possible, avoid DB specific SQL extensions so as to ensure cross-database portability

Stored Procedures

Oracle PL/SQL conversion is a little problematic and the obvious PostgreSQL backend language in which to (re)write stored procedures is the similar procedural language PL/pgSQL.

To install PL/pgSQL, the superuser DBA should run,
        $ createlang -d db9 plpgsql   *# install 'Oracle PL/SQL like' language*

where db9 → database

Concluding Remarks

This brief paper demonstrates, for R&D/information purposes,   some of the changes required for converting a J2EE application from using Oracle 10.2 to working against PostgreSQL 8.1 & 8.2.4.

Chris Drawater has been working with RDBMSs since 1987 and the JDBC API since late 1996, and can be contacted at *chris.drawater@ericssonservices.co.uk* or *drawater@btinternet.com* .

References

1. (Drawater 2006), PostgreSQL 8.1 for J2EE/JDBC applications v1.0 @ http://www.holindis.co.uk/