Author : Chris Drawater
Date : 10/01/2002
Updated : 15/03/2007
Version : 1.1

A Design Technique for scaling Java Middleware – Oracle systems.

Abstract

Holistically designing middleware Java applications and Oracle databases as a single entity , can enable massive transaction scalability across multiple databases and hardware platforms, without application change.

Document Status

Introduction

Many internet or mobile application systems use the now classical multi- tier model of
*n * clients – n * Web/Application Servers – Oracle Database*
and most, if not virtually all, systems are designed to work against a single DB on a single node, or at best, a single database running under RAC.

The single database is potentially the least scaleable system component in terms of user transaction throughput, and so problems arise when database performance cannot scale over the single hardware node or cluster.

This paper explores one (of many) holistic techniques for building massively scaleable middleware/server systems using Oracle and Java technologies, although this design blueprint can equally be applied to other databases such as PostgreSQL.

Pseudo-code Java examples derived from production code illustrate the technique.

Abbreviations & Definitions

RAC → Oracle Real Application Cluster
DB → Oracle database
VLDB → Very Large DB
JTA → Java Transaction API
TX → Transaction
AS → Application Server
OS → Operating System
HA → High Availability

Concept

So how do we go about scaling a system ? The essential concept in the ability to partition , and later re-partition again, data across databases and navigate to that data (without modification of application code).

The basic steps are as follows →

(1) At database level ,

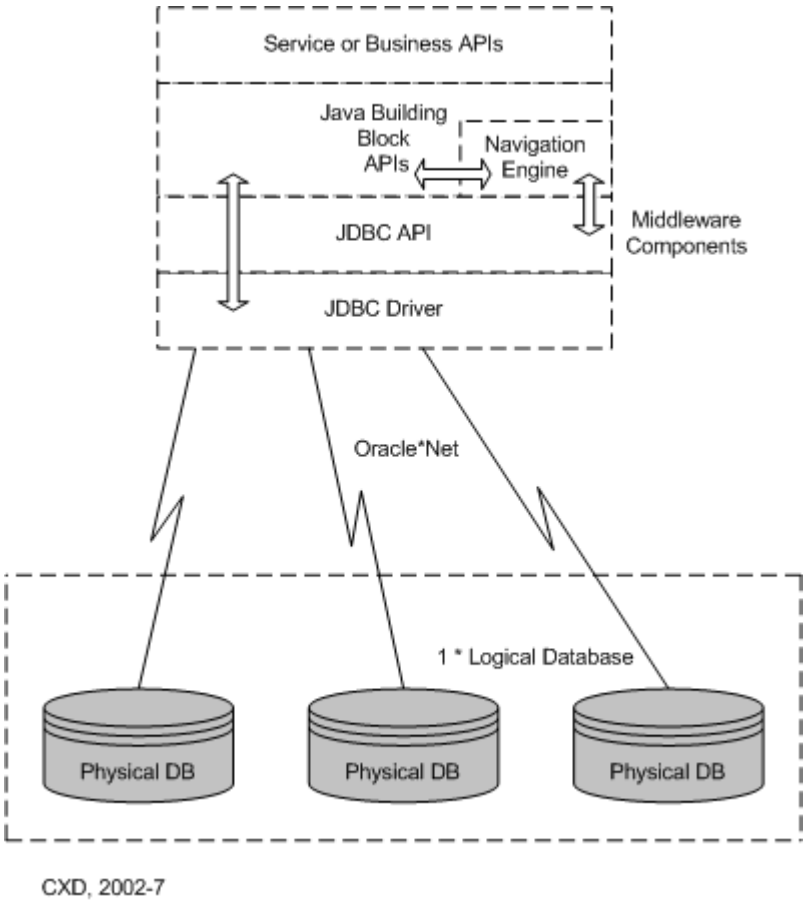
- Consider your database to be a logical database implemented as a number of physical databases.
- Design the DB tables to be accessed by a key. For example, for the purposes of this paper, customer account data might be keyed by customer login.
- Partition your data by key across multiple DBs (lexicographically is easiest). For customer account data, let's say DB1 hosts customer logins beginning 'A' through to 'J' (ie. lexicographically less than 'K') and DB2 holds range 'K' to to a conceptual but undefined MAXVALUE. We could have used more than 2 databases but the principle is the same. The Oracle databases can be deployed upon standalone hardware nodes and/or hardware clusters supporting RAC or HA failover.
- If data such as reference information needs to be available to each DB, consider replication rather than locating centrally.

(2) At middleware level,

- Build a Java navigation engine class to control routing of JDBC access to the Oracle databases.
- Encapsulate your SQL into a series of (Java) "building block" APIs - these application building blocks dynamically interact with the navigation engine to deliver SQL requests to the correct database. A side benefit of doing this is that it enforces the use of your (hopefully) highly optimised SQL access to your databases and so squeezes more TX/Sec out of your system and so delays the need to scale up…
- Where possible, build business functions as single DB interactions. Calls distributed across more than 1 database and coordinated as a single atomic TX are possible but more of that later.
- If possible, ensure that your 'client to middleware' calls are effectively stateless by using cookies, URL rewrites or simply state held in DB (again against key) . Having stateless/sessionless middleware applications will allow any client to connect to any of a number of web-middleware servers so making the middleware layer more scaleable. A bonus is that this will also permit the 'mix and match' of Web Servers & J2EE engines, even different OS and/or hardware platforms (NT, Linux, Solaris) if so required.

Our basic architecture is shown in Figure 1. Now onto the implementation in a little more detail…

Figure 1 Basic Partition-Navigation Architecture



Database Schema

A common misconception is that Oracle level partitioning is only of use with VLDB such as datawarehouses. Not so ! Partitioning OLTP data makes for easier database maintenance, shorter periods that tablespaces are kept in online backup mode, together with a finer granularity of query optimiser statistics.

Continuing our conceptual example, each of our databases DB1 and DB2 can contain tables that themselves can be partitioned.

If DB1 contains account data in the key range 'A' to 'J' (less than 'K'), we could range partition the table as follows →

```
create table eAcct
(
        cust_reference     varchar2(100),
        …….
)
storage ( buffer_pool default)                          --  all  tablespace/segment space mgmt auto
noparallel
partition by range (cust_reference)
        (
        partition p1 values less than ('D') tablespace eAcctdata1,          -- eg A,B,C
        partition p1 values less than ('G') tablespace eAcctdata1,          -- eg D,E,F
        partition p3 values less than ('I') tablespace eAcctdata3,          -- eg G,H
        partition p4 values less than (MAXVALUE) tablespace eAcctdata4      -- eg. I.J  (ie < K)
        );
```

This partitioning provides us with discreet subsets of data that can be individually shipped or even transported directly as tablespaces to other databases. For example we could move the 3rd and 4th partitions (effectively key range G to J) to a new database DB3.

This would leave our logical database physically implemented with key ranges :

DB1 : A,B,C,D,E,F (< G)
DB3 : G,H,I,J (< K)
DB2 : K-> Maxvalue

The ability to easily subdivide and move data is one of the keys to the scalability of the logical database .

Another key is the ability to navigate to that data.

Java Navigation Engine

A navigation engine class, such as shown in listing 1, needs to be able to

- Create or acquire a pool of JDBC *Connection* Objects
- Route access to each DB via its *Connection* object according to our chosen partitioning key

In our pseudo code example *vDBNavigator* , the *Connection* objects relating to each Oracle database are held within the *vDBConnection* class (Listing 2) and are set up in the constructor method. Using key range & Oracle connection information held within a Java properties file (such as in Listing 3), a *Vector* of *vDBConnection* objects in order of range keys can be constructed. The *getOraConnection()* method acquires the actual Oracle database connection.

Within our Java "building block" classes, all database access against our chosen partitioning key (customer account name) is routed using the method *findDB()* which returns the Vector element that holds the appropriate DB Connection.

Most importantly, the lexicographical sort order of the database MUST match that of your Java code for the navigation engine & partition splits to function correctly.

Application

Within the constructor method of our example "building block" class(Listing 4), the *vDBNavigator* object is instantiated (or better still will have been passed across so as to reuse Oracle connections), and any *preparedStatement* objects created for each database.

In the methods implementing our business functions , assuming the navigational key is known, the correct database connection can be obtained as in the *getAcctDetails()* method . The *findDB()* method of the *vDBNavigator* object works through its vector of *vDBConnection* objects , comparing our navigational key with the max key values until a match is determined.

For non-prepared *Statements* we could simply obtain the *Connection* object and use that directly.

There is no hardcoded navigational access to any database, therefore no code changes are required when the underlying database configuration is modified.

Distributed Transactions

So far, we can navigate to our databases and execute SQL and commit local database transactions.

But what happens if we wish to deploy a business transaction that writes to 2 or more databases. ?
Well, we have 2 basic options →

- We can utilise a psuedo distributed TX with a 'Retry on fail' protocol – these are not real distributed TX using XA and 2 phase commit. Instead, the functional or business TX is comprised of a sequentially executed series of individual DB TX . These individual DB TXs can be re-run without adverse consequence and their sequence of execution runs from least crucial through to most important.

- However, there are a few scenarios where a true atomic distributed TX is required, for example, when posting financial transactions. Enter JTA !. In addition, to tracking URL/JDBC *Connection* objects against key range in our *vDBConnection* class, a JDBC *DataSource* object could also be held. The method of navigating to the correct database is as before but this time the *Connection* objects are obtained via the *DataSource* object and used within the scope of a *UserTransaction* object. Note that the AS must support JTA global transactions else you should use the Oracle database itself as the (JTA) transaction manager.

Scaling

So what happens when one of our hardware nodes finally runs out of steam, and we need to increase the number of databases/nodes ? The process is as follows →

- Switch the tablespaces containing the data to be moved into 'read only' mode
- Move data to the new DB
- Modify the data partitioning/navigation configuration to incorporate the new DB and propagate to the middleware modes
- Reboot the middleware servers to pick up new configuration – no application changes required!
- Archive off the original data (now in read only mode) & re-use space

Issues

Every architecture has its trade-offs and with this data partition-navigation model, there are a few (but not insurmountable) issues→

- The application must be based around 1 or 2 navigational data entities.
- Non-navigational access to DB partitioned data may need to make calls to all underlying physical nodes – avoid where possible!
- Consolidating data from more than 1 database into a single dataset must be handled within the application (for a limited number of rows) or more rarely, via a (carefully crafted and tuned) database level distributed select..
- Implementing Distributed TX (as discussed above) is a little more complex.
- Sequence numbers etc must be global (ie. unique across all underlying physical databases)
- The lexicographical sort order of the database MUST match that of your Java navigation class.

Concluding Remarks

Holistically designing middleware Java applications and Oracle databases as a single entity (rather than as 2 system components in isolation) can allow us to design for massive transaction scalability across multiple databases and hardware platforms, without application change.

This technique has been successfully used within the telecommunications industry , but it is not the only way to scale applications. For example, Oracle 10g grids potentially offer massive scalability but themselves have their own issues (for example, cross node locks, interconnect latency etc).

This technique is merely one of many that should reside in your toolkit!

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

Listing 1

```
public class vDBNavigator
{
        public Vector wlist;                    // of vDBConnection objects
        public vDBConnection singleDBConnection;
        private int NO_DBS = 10;                        // extract value out of config file
        private Connection con = null;

        public vDBNavigator() throws Exception
        {
                // obtain NO_DBS out of config file

                for (int DBi = 0; DBi < NO_DBS; DBi++)
                {
                        // Obtain  Oracle connectioninfo & , maximum key value out of config file
                        // get Connection
                        con =  getOraConnection(...);

                        singleDBConnection =  new vDBConnection(DBi, max_key,...,con);

                        // add to vector
                        wlist.addElement((Object)singleDBConnection);
                }
        }

        private Connection getOraConnection(...) throws Exception
        {
                Connection con;
                ....
                return con;
        }

        public int findDB(String key) throws Exception
        {
                vDBConnection c;
                int vLen = wlist.size();
                for (int i=0; i < vLen; i++)
                {
                        c = (vDBConnection)wlist.elementAt(i);
                        if (key.compareTo(c.max_key) < 0 )
                        {
                                        return c.ientry;
                        }
                        if (c.max_key.equals("MAXVALUE"))
                        {
                                        return c.ientry;
                        }
                }
                // if reached this far then raise exception
                throw new Exception("vDBNavigator.findDB() : cannot locate DB for " + key);
        }
} // end vDBNavigator class
```

Listing 2

```
public class vDBConnection
{
        public int ientry;
        public String max_key;
        ...
        public Connection con;

        public vDBConnection (int ientry, String max_key,..., Connection con)
        {
                this.ientry = ientry;
                this.max_key = max_key;
                ...
                this.con = con;
        }
}
```

Listing 3

```
################################################
# PARTITIONING - do NOT modify
################################################
#
NO_DBS=2
w0.boundry=K
w1.boundry=MAXVALUE
#
################################################
# JDBC CONNECTION INFO
################################################
#
# DB1
w0.primary.url=jdbc:oracle:thin:@CXDSUN1:1521:db8
w0.primary.url_uname=cxd
w0.primary.url_passwd=XXX
#---------------------------------------
# DB2
w1.primary.url=jdbc:oracle:thin:@CXDSUN2:1521:db9
w1.primary.url_uname=cxd
w1.primary.url_passwd=XXX
################################################
```

Listing 4

```
public class myJavaAPI
{
        private vDBNavigator  dbpool;       // DB navigation object
        private Connection con;
        private PreparedStatement getAcct[] = new PreparedStatement[MAXDB];

        public myJavaAPI() throws Exception
        {
                // Navigation Engine
                dbpool = new vDBNavigator();

                // Setup prepared staremements against all Oracle connections
                for (int i = 0; i < dbpool.wlist.size(); i++)
                {
                        con = ((vDBConnection)dbpool.wlist.elementAt(i)).con;
                        getAcct[i] = con.prepareStatement("select…. where acct = ?");
                }
        }

        public getAcctDetails(String acctname) throws Exception
        {
                int i = dbpool.findDB(key);
                getAcct[i].setString(1,acctname);
                ResultSet rs = getAcct[i].executeQuery();
                ….
                rs.close();
        }

} // class myJavaAPI
```