

# DBMS\_JOB and DBMS\_PIPEdreams

Douglas M. Scherer  
Core Paradigm

## Introduction

My colleague and I were working within a very aggressive schedule. Our mission was to set up a nightly load from an external proprietary database into an Oracle database production system. We were working in a UNIX environment, so we could have used cron to coordinate the nightly batch jobs, but we wanted our solution to be independent from the operating system. We might want to port to another OS in the future. Using cron jobs would also have required us to get the systems administration staff involved, and we preferred to look for a solution with more independence - our machine was dedicated to our database, and we naturally wanted to operate as autonomously as possible.

After some experimentation and fine tuning, we came up with a solution that has been running for the past nine months. We used a combined application of Oracle's DBMS\_JOB and DBMS\_PIPE packages along with a set of C programs to perform the following totally within Oracle.

- kick off the nightly job automatically on a periodic basis from within the database
- coordinate PL/SQL code with C programs running in the OS
- manage job flow (i.e. job dependency tree)
- keep a log of step by step job completion along with any errors that might have occurred
- call OS code on the server, that we typically use for the nightly batch run, from any client front end
- start or restart the nightly batch runs from the beginning or any predetermined midpoint
- simplify the nightly tasks of the Operations group

move the job queue and code from the development database to the production database with ease

This paper briefly describes DBMS\_JOB and DBMS\_PIPE, describes how we used them, and provides an experiment you can run to experience some of this functionality first hand.

## Why We Used DBMS\_JOB

DBMS\_JOB is an Oracle provided package that allows you to run stored procedures, functions and packages at specified times and intervals.

DBMS\_JOB is being used in lieu of cron or another scheduler. By avoiding cron we avoid coupling the batch job to the cron facility. This independence from UNIX's cron will have an added payoff if we ever move to another operating system. With this setup we can use the export utility to dump a user's schema and use the import facility to load any jobs they've set up in the Oracle job queue.

## Why We Used DBMS\_PIPE

We used DBMS\_PIPE for two purposes:

- to send commands to a C program running in the OS
- to log status messages into an Oracle table in a separate session/transaction than the one issuing the status message

## Using DBMS\_JOB

In order to use DBMS\_JOB the database must be started with some additional parameters in the init.ora file. The three parameters specify:

- **JOB\_QUEUE\_PROCESSES**: the number of snp processes that will start with the database
- **JOB\_QUEUE\_INTERVAL**: the number of seconds between each snp process wake-up call
- **JOB\_QUEUE\_KEEP\_CONNECTIONS**: whether or not remote database connections should be closed after all the jobs have completed

The range for **JOB\_QUEUE\_PROCESSES** is 0 to 10, with a default of 0. Oracle Technical Support recommends using between 2 and 5. An Oracle Support Technician told me never to specify more than 5, and that there is no way of determining how many snp processes you'll need. Both **JOB\_QUEUE\_PROCESSES** and **SNAPSHOT\_REFRESH\_PROCESSES** parameters specify the number of snp processes. In a test environ-

ment of SunOS 4.1.3 and Oracle 7.1.6 the SNAP-SHOT\_REFRESH\_PROCESSES value took precedence over JOB\_QUEUE\_PROCESSES.

To specify the number of seconds between each snp wake-up call, use the JOB\_QUEUE\_INTERVAL parameter. The range is 1 to 3600, with a default of 60.

For example, to specify every 15 minutes use: JOB\_QUEUE\_INTERVAL=900.

The range for JOB\_QUEUE\_KEEP\_CONNECTIONS is TRUE or FALSE, with a default of FALSE.

### Managing Oracle's Job Queue

It's simple to submit and remove jobs from Oracle's job queue. There are examples of these two operations in the Experiment section of this paper. Look for the calls to DBMS\_JOB.SUBMIT and DBMS\_JOB.REMOVE. Killing an active job is somewhat of a nightmare. The procedure as shown below, seems simple enough, but this never seems to go smoothly.

1. Mark the job as broken(as shown in the example below). The job queue will not run a job as long as it is marked broken.

```
•exec dbms_job.broken(5,TRUE)
```

2. Kill the session that's running the job. You may need the DBA to do this for you.

### Warning for AIX Users

Be aware of bug 267941 in AIX. It causes the alert.log file to grow beyond earthly reason (at least 100s of MB) if an error occurs in relation to the job that got kicked off from the queue (e.g. if the DBMS\_JOB attempts to run a procedure which has since been invalidated). There is a patch available for this. The alert.log is used for these errors since DBMS\_JOB is tightly tied to the snapshot mechanism. It would be nice if Oracle provided the ability to specify an alternate log file to the alert.log.

### Using DBMS\_PIPE

To send a message, use DBMS\_PIPE.PACK\_MESSAGE and DBMS\_PIPE.SEND\_MESSAGE. To receive a message from a pipe use DBMS\_PIPE.RECEIVE\_MESSAGE and DBMS\_PIPE.UNPACK\_MESSAGE. The examples in the Experiment section of this paper will clarify how to use this functionality.

### Putting it All Together

This section describes a simplified version of what we do on a nightly basis. In your environment you will probably find that it will take some extra work to get it to gel. However, once you've gotten the infrastructure set up, it's a fairly straight forward process to add or remove jobs form the nightly batch run. The rest hap-

pens while you're sleeping. We encountered some glitches during the first week or two that the system was in production, most of these revolved around the timing of events in coordination with the external proprietary database, and were easily fixed.

We adhered to the following guidelines:

- the OS C programs will be used as much as possible for the sole purpose of placing external data into a transition/holding table in Oracle (we call this a PULL)
- the PL/SQL programs will be used as much as possible for programs which move the data from the transition table into the final production tables (we call this a LOAD)

### The Overall Scenario

- To simplify the description, assume that all sessions connected to the Oracle database are connected to the same account name.
- A job in the Oracle job queue starts a stored procedure named JobQueuePullAndLoad at 2:00 AM each morning.
- JobQueuePullAndLoad initiates the PULL - that is, it puts the data in the transition table from the external database - by sending a message via a database pipe (P1) to a C program (C1). The message tells C1 to start the C program responsible for the PULL, that is, C3.
- Immediately after sending the message, JobQueuePullAndLoad waits for a message via a database pipe (P3) telling it that C3 has completed the PULL.  
This ensures that JobQueuePullAndLoad will always wait until C3 is done placing data in the transaction table, whether this takes 5 seconds or 5 hours.
- C1 gets the pipe message from JobQueuePullAndLoad to start C3, kicks off C3, and resumes waiting for another pipe message.
- While C3 is PULLing data and building the transition table data, it sends status messages reporting its progress at various stages to another C program (C2) via a database pipe (P2).
- C2 writes the status information to a table called MESSAGE\_LOG. C2 is used for this job so that the committing of messages to MESSAGE\_LOG will not interfere with ongoing transactions occurring in the program that issued the message.
- When C3 is done building the transition table data, it does two things in this order:

- It sends a message to C2 via a pipe (P2) stating whether or not it's PULL efforts ended in success or failure.
- it sends a message to JobQueuePullAndLoad via the pipe (P3) stating that the PULL effort is finished.
- C2 writes the success or failure status of the PULL to MESSAGE\_LOG.
- JobQueuePullAndLoad receives the completed message from C3 via the database pipe P3, that the data's been put into the transition table.
- JobQueuePullAndLoad checks MESSAGE\_LOG to see if C3's effort to PULL was successful.
- Once JobQueuePullAndLoad determines that C3 completed the PULL successfully, it runs the PL/SQL program that LOADS the data from the transition table to the production tables.

### Some Additional Pieces to the Puzzle

- The Operations group can monitor C3's progress through an Oracle form that looks at the MESSAGE\_LOG table.
- We issue TRUNCATE with reuse storage option on the transition table before each PULL.

## Experiment

Here's where you get to try some of this stuff. We'll use a simple experiment that you can set up in your own database. The experiment will make use of two tables (one with a trigger on it), the DBMS\_PIPE package, the DBMS\_JOB package and a stored package. We'll assume an account name of OPSSMYSCHEMA with connect and resource privileges.

1. Make sure that the database was started with the init.ora parameters as described above.
2. Logon to Oracle via SQL\*Plus as OPSSMYSCHEMA.
3. Run the following create table statements:

```
create table employee
(id number
 constraint employee_id_nnull not null,
 salary number
 constraint employee_salary_nnull
 not null,
 constraint employee_pk primary key
 (id)
 using index
 );
create table message_log
(message_date date,
 message varchar2(100)
 );
```

### 4. Run the following CREATE PACKAGE statement:

```
create or replace package PipeDream
as
procedure AbortMessageHandler;
procedure GetLogMessage;
procedure InsertLogMessage
(InMessage message_log.message%type);
procedure SendMessageToLog
(InMessage message_log.message%type,
 InUser user_users.username%type default user);
procedure StartMessageHandler;
TheBossSalary constant number := 1999999;
end PipeDream;
```

### 5. Run the following CREATE PACKAGE BODY statement:

```
create or replace package body PipeDream
as
procedure AbortMessageHandler
-- This program will cause the GetLogMessage
-- program to stop running.
is
begin
SendMessageToLog('ABORT');
end AbortMessageHandler;
procedure GetLogMessage
-- This program waits on a database pipe for
-- a message. If it receives a message that
-- says "ABORT" it will stop running. Any
-- other message it receives will be inserted
-- into the MESSAGE_LOG table.
is
PipeReturnVal number;
LocalMessage message_log.message%type;
ProcName constant varchar2(60) :=
'PipeDream.GetLogMessage';
begin
loop
-- Loop until an "ABORT" message is received.
-- Start waiting for a message from the data
base pipe
-- called P2.
PipeReturnVal:= dbms_pipe.receive_message( 'P2');
if PipeReturnVal = 0
then
-- message was received successfully
dbms_pipe.unpack_message(LocalMessage);
if LocalMessage != 'ABORT'
then
-- A message has come through the pipe
-- destined for MESSAGE_LOG. Insert
-- the message.
InsertLogMessage(LocalMessage);
else
-- An ABORT message was received. Put a
-- message in the log that an ABORT is
about to
-- to occur, then exit.
InsertLogMessage(user||': '||ProcName||
' ABORT requested.');
```

```

        PipeReturnVal);
    exit;
end if;
end loop;
exception
when others
then
    -- Some other error has occurred in this
    program. Report
    the error in MESSAGE_LOG then exit.
    InsertLogMessage(user||': '||ProcName||
        ' '||substr(sqlerrm,1,50));
end GetLogMessage;
procedure InsertLogMessage
(InMessage message_log.message%type)
is
    -- This program receives a message to put into
    MESSAGE_LOG
    -- then inserts it.
begin
    insert into message_log
        (message_date, message)
    values (sysdate, InMessage);
    commit;
exception
when others
then
    raise_application_error(-20004,'Error:
||sqlcode||
    'occurred in InsertLogMessage.');
```

```

end InsertLogMessage;
procedure SendMessageToLog
(InMessage message_log.message%type,
InUser user_users.username%type)
is
    -- This procedure handles sending a message,
    destined
    -- for MESSAGE_LOG into the P2 pipe. At
    the other
    -- end of the pipe, a program will be wait-
    ing for the
    -- message and will commit it to MESSAGE_LOG
    in
    -- a separate transaction/session.
LocalMessage message_log.message%type;
PipeReturnVal number;
begin
    if InMessage != 'ABORT'
    then
        -- Tack the current user's name onto the
        message.
        LocalMessage := InUser||': '||InMessage;
    end if;
    -- Send the message into the pipe called P2.
    dbms_pipe.pack_message(LocalMessage);
    PipeReturnVal := dbms_pipe.send_message('P2');
    if PipeReturnVal != 0
    then
        -- An error occurred while sending the mes-
        sage into P2.
        -- Report the error to the application.
        raise_application_error(-20001,
            'PipeDream.SendMessageToLog failed with
            Pipe error: '||
                PipeReturnVal||'.');
```

```

    end if;
end SendMessageToLog;
procedure StartMessageHandler
is
    -- This program starts an event handler - here
    -- called GetLogMessage in its own session.
```

```

JobNumber number;
begin
    dbms_job.submit(JobNumber,'PipeDream.
    GetLogMessage;');
    commit;
end;
end PipeDream;
```

## 6. Run the following create trigger statement:

```

create or replace trigger before_employee_iu
before insert or update on employee
for each row
declare
    ErrorMessage constant message_log.message%type
:=
    'New Salary was too high for employee # ' ;
begin
    if :new.salary > PipeDream.TheBossSalary
    then
        -- An attempt has been made to raise the
        -- employee's salary to a rate greater than
        -- that of the boss's salary. Send a message
        -- to MESSAGE_LOG that this attempt has been
        -- made, then force the transaction to roll
        -- back.
        PipeDream.SendMessageToLog(ErrorMessage||:new.id);
        raise_application_error(-20010,
            ErrorMessage||:new.id);
    end if;
end before_employee_iu;
```

## 7. Issue the following statement at the SQL\*Plus prompt:

```
exec PipeDream.StartMessageHandler
```

## 8. Check to make sure that GetLogMessage is running.

### • Issue the following from the SQL\*Plus prompt:

```

select job, this_date, this_sec
from user_jobs
where what = 'PipeDream.GetLogMessage;'
```

### • If this\_date and this\_sec are null, then the snp process has probably not yet woken up. Wait for the same number of seconds as specified in the JOB\_QUEUE\_INTERVAL and run the select statement again.

## 9. insert a valid row into EMPLOYEE and commit it.

## 10. Start a second SQL\*Plus session, connecting as OP\$MYSCHEMA.

### • You'll use this account to check MESSAGE\_LOG

## 11. Issue the following statement:

```
select * from message_log;
```

### • You'll find that there are no status messages listed.

## 12. In the initial SQL\*Plus session insert a row into EMPLOYEE where the salary is greater than the boss's salary (200000 for example)

- update employee set salary = 2000000;
- do not issue a commit

13. Return to the second SQL\*Plus session and select \* from message\_log. You'll find that there's an error message in the log from the uncommitted and invalid transaction of the initial session.
14. Remember to shut down the message handler when you're done experimenting. You do this by issuing the following from the SQL\*Plus prompt:

- exec PipeDream.AbortMessageHandler

## Conclusion

The usefulness of this approach may not at first be apparent, but when you consider the possibilities of using it to provide an event handler for your application inside the Oracle database you can see that it becomes a very powerful tool. You can use it to develop applications like a job queue facility.

## Acknowledgments

Fritz Cloninger - Editorial assistance

Chris McKeon - Co-analysis/Design and C programming

## Additional Readings

Bobrowski, Steven M., Oracle7 & Client/Server Computing. San Francisco: SYBEX, 1994

Feuerstein, Steven, ORACLE PL/SQL. Sebastopol: O'Reilly & Associates, 1995.

McPherson, Andrew. "Oracle Job Queue Facility." SELECT (January 1996) 39-42.

Oracle, Oracle7 server application developer's guide: release 7.2. Redwood City: Oracle Corporation, 1995.

Pratt, Mary. Oracle7 server distributed systems: replicated data: release 7.1. Redwood City: Oracle Corporation, 1995.

Oracle Scripts dbmsjob.sql and dbmspipe.sql