Fast Oracle Parallel Exports on Sun Enterprise™ Servers

By Stan Stringfellow - Enterprise Engineering

Sun BluePrints™ OnLine - March 2000

http://www.sun.com/blueprints

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax 650 969-9131
Part No.: 806-4980-10
Revision 01, March 2000
Fast Oracle Parallel Exports on Sun Enterprise™ Servers

This article is derived from the Sun BluePrints™ book titled Backup and Restore Practices for Sun Enterprise™ Servers by Stan Stringfellow, Miroslav Klivansky, and Michael Barto. The book is a practical guide for IT organizations that are tasked with implementing or revamping a backup/restore architecture. It includes case studies, a methodology, and example runbooks. It addresses issues such as scalability and performance of the backup/restore architecture, criteria for selecting tools and technologies, and tradeoffs that must be considered. It provides technical guidelines for planning the architecture to meet service levels, as well as general advice and guidance. For more information, see http://www.sun.com/books/blueprints.series.html.

Why a Parallel Export Script?

This article describes a script that performs very fast Oracle database exports by taking advantage of parallel processing on SMP machines. The performance benefit that this script provides can be nearly linear with the number of CPUs. For example, if you have 10 CPUs available for backups, you may be able to achieve nearly 10 times the normal database export speeds. Thus, this script can be extremely valuable for situations where you need to perform exports of large mission-critical databases that must be highly available.

The script was kindly contributed by its authors, Neal Sundberg and Mike Ellison of QUALCOMM Incorporated. At QUALCOMM, the IT staff routinely runs this script on Sun Enterprise servers to perform an export of a 250 GB database in about two hours. That is very hard to do without using parallel processing on SMP machines.
The Oracle utilities for export and import are basically single-threaded processes that use a single CPU, with the exception of the Oracle feature called parallel query. Under certain circumstances, the parallel query feature enables you to spawn parallel processes that perform a SELECT against a large table. You can set the degree of parallelism on the table to the desired number of processes, such as 4 or 8. The parallel query feature takes advantage of multiple CPUs on an SMP machine, possibly breaking down the SELECT statement into separate smaller SELECT statements. However, you might not want to apply parallelism to all of your tables since, in some cases, this can cause problems.

Oracle provides features that support physical backup/restore as well as logical backup/restore. The physical backup/restore features are the normal hot and cold backups, where you simply take static images of the database files (although the hot backups are not quite static). Basically, you copy the database files without really knowing what’s in them. With this approach, if you lose a disk drive or if a controller writes bad bytes onto the database, you can recover from the database point of view. You really don’t care what is in the tables.

Export/import, on the other hand, takes the logical point of view. An export contains all of the commands required to recreate the database, as if those commands were processed through the Oracle engine. For example, it includes commands such as CREATE DATABASE, ADD DATA FILES, and CREATE TABLE, as well as commands to add data to tables. Thus, an export/import is similar to redoing the entire database by typing everything in by hand.

The export/import approach can be useful for migration, for example from one operating system to another. It can also be useful if you simply want to logically restore part of a database that was lost. Perhaps you only lost one or two tables, and you simply want to restore those tables. You can’t use the physical methodology in this situation unless those tables happen to be in their own table space, which is unlikely. There are cases where you might want to use export/import to rebuild your entire system. Maybe you want to defragment your database. This is probably the only way to do it, unless you buy a third party tools.

It is also possible that the export/import approach can supplement normal hot backups, and may sometimes be a lifesaver. For example, suppose your disk RAID failed, and your hot backup failed. In this case, you would be very fortunate if you had a database export available to you, since it could serve as another layer of protection against failure in your system and could allow you to recreate the database. However, an export is only good up to the time it was taken; you can’t apply transaction logs. (This is because transaction logs in Oracle use system change numbers, or SCNs. The SCNs help to maintain consistency with the rest of the system. But, SCNs are specific to a particular instance of the database. Since, with export/import utility you are rebuilding the database, the SCN numbers have no significance.) But, missing a day worth of transactions is far better than having no data at all! You probably wouldn’t keep database exports around for system failures, except as a third or fourth line of defense, depending on your configuration. However, as mentioned, exports are useful for logical restores.
Consider a situation where a new software program is deployed, and due to a lack of thorough testing, the new program corrupts some tables in your database. You might want to avoid rebuilding the entire database, and simply restore those tables that were affected by the new program. In this case, it would be extremely desirable to have an export image that you could perform imports against. There are other ways you could recover from this situation, but the export/import method may be the fastest. For example, you could rebuild the entire system on the test environment, and bring it up to the point in time just before the failure occurred. You could then open the test system, and export the tables from there. Then, you could use those exports to perform imports into the production system. However, this approach assumes that you have a test system that can be down for a while, and that you have the luxury of the extra time required to go through this entire process.

Exports can be taken in either of two ways. You can do a hot export, with database up and running, and transactions occurring. But, in this case, the backed up tables might be read-inconsistent with each other. So, you could use this approach if you wanted to recover only one table, or perhaps multiple tables if you are willing to perform some SQL+ re synchronization. The second way to do an export is to take the database down and put it into restricted mode (which used to be called DBA mode). Since all tables are consistent, you can completely restore the database from this export.

QUALCOMM does not typically use exports for full database creates, or even as a recovery strategy. But, it can be three layers deep in the recovery strategy. The first layer would be protection from a disk failure, or a hardware failure. This is done with fault tolerant systems, RAID, and so forth. The second layer of protection is an Oracle hot backup. This is the familiar method of putting the database into archive log mode, and then backing up the archive logs between full backups. In this way, if you experience a media failure or a physical system failure, you can recover the entire database from the full and incremental backups on tape. However, in the event that there was a problem with those tapes or the data on them, the export can serve as a last line of defense.

**Overview of the paresh Approach**

The main script used to implement Oracle parallel exports is called paresh, for “Parallel Export Shell”. (Also, the author of the script once worked with a consultant from India named Paresh Shah.) The basic approach involves two steps:

1. Take a structure-only export of the entire database. This includes object definitions; objects being tables, indexes, clusters, and so forth. It also includes users, accounts, and so forth. However, no data is exported at this point. This step usually takes about 15 to 20 minutes depending on the size of the database, the
hardware that is used, and so forth. Build a list of database export commands
based on this information. To perform this step, you might use an approach such
as the one described in “The list_tbls Script” on page 5.

2. Export all of the data, one table at a time. There might be thousands of tables in
the database. The paresh script keeps a specified number of table export
processes running at any given time. The paresh script spawns processes for
each of the export commands that were defined in Step 1, above. The paresh
script is described in more detail under the heading, “The paresh Script” below.

This approach uses a master process that spawns as many slaves as you ask it to. For
example, if you set the level of parallelism to 10 processes, the paresh script takes
the top 10 items in the list of tasks that was produced by the list_tbls script, and
spawns those 10 processes. As each process completes, paresh takes the next item
from the list and spawns a new process for that item. In this way, it keeps 10
processes running at all times, until the entire database export is completed. This
would make optimal use of a backup server with 10 CPUs.

Note – The paresh script is basically a traffic cop for doing “parallel anything”.
You could conceivably create a completely different set of tasks than those created
by the list_tbls script, and still make good use of paresh to spawn those tasks in
an optimal fashion.
The list_tbls Script

This SQL*Plus script serves as an example of how to create a file of commands that can be used by the paresh script. There are many different approaches. The list_tbls.sql script references the standard scripts table_export.sh and full_export_no_rows.sh. The only parameter to list_tbls.sql is the file name of the file to be used by the paresh script.

```sql
whenever sqlerror exit sql.sqlcode;
set pause off;
set pages 0;
set linesize 132;
set feedback off;
set termout off;
column cmd_line format a80
column tick format a3
column sum_bytes format 999,999,999,999
!/bin/rm -f &&1;
spool &&1;
select './full_export_no_rows.sh' from dual;
select './table_export.sh '||owner||' '||segment_name cmd_line,
' # ' tick,
sum(bytes) sum_bytes
from dba_segments
where segment_type = 'TABLE'
and owner <> 'SYS'
group by owner, segment_name
order by sum(bytes) desc
;
spool off;
exit;
```

The paresh Script

This shell file is used to control multiple shell files, in a parallel fashion. It takes two arguments, a file name and the number of processes to run concurrently. The input file is expected to contain a list of all commands to be executed, along with their arguments. The number of parallel processes defaults to 4, but can be set to any positive value.
The usage is:

```
# paresh commands_file parallel_count
```

where:

`commands_file` is a file that contains a list of commands or shell files to be executed. Because all processing occurs in parallel, there is no guarantee of what the order will be, although the commands are processed from the beginning of the list to the end.

`parallel_count` is the number of slave processes that should be active simultaneously.

```
#!/bin/sh
#
#-------------------------------------------------------------
# message
# Establish a timestamp and echo the message to the screen.
# Tee the output (append) to a unique log file.
#-------------------------------------------------------------
#
message()
{
    timestamp=`date +"%D %T"`
    echo "$timestamp $*" | tee -a $logfile
    return
}
```
#-------------------------------------------------------------
# get_shell
# This function is responsible for establishing the next
# command to be processed. Since multiple processes might
# be requesting a command at the same time, it has a built-
# in locking mechanism.
#-------------------------------------------------------------

get_shell() {
    echo "'date' $1 Shell Request $$" >> $lklogfile
    # debug locking file
    # until a command or end
    while :
        do
            next_shell="" # initialize command
            if [ ! -s ${workfile} ] # if empty file (end)
                then # no more commands
                    break
                fi #
            if [ ! -f $lockfile ] # is there a lock?
                then # not yet...
                    echo $$ > $lockfile # make one
                    echo "'date' $1 Lock Obtained $$" >> $lklogfile
                    #debug
                else # variable "next_shell"
                    echo "'date' $1 Lock FAULTED $$" >> $lklogfile # debug
                    fi # double check faulted
            if [ "$$" = "'cat $lockfile'" ]
                then # double check that
                    next_shell='sed -e q $workfile'
                    # first line of file
                    sed -e 1d $workfile > ${workfile}.tmp # Chop 1st line
                    mv ${workfile}.tmp $workfile # rename to work file
                    rm -f $lockfile # turn off lock
                echo "'date' $1 Shell Issued " >> $lklogfile #debug
                return # done, command in
            else # locked by other
                echo "'date' $1 Lock Wait $$" >> $lklogfile # debug
                fi # try again
            sleep 1 # brief pause
        done # try again
    return # only if no commands
}
#-------------------------------------------------------------
# paresh_slave
# This code is executed by each of the slaves. It basically
# requests a command, executes it, and returns the status.
#-------------------------------------------------------------
# paresh_slave()
{
  shell_count=0 # Commands done by this slave
  get_shell $1 # get next command to execute
  while test "$next_shell" != ""
    # if no command, all done
    do # got a command
      shell_count=`expr $shell_count + 1`
        # increment counter
      message "Slave $1: Running Shell $next_shell"
        # message
      $next_shell # execute command
      shell_status=$? # get exit status
      if [ "$shell_status" -gt 0 ]
        # on error
        then # then message
          message "Slave $1: ERROR IN Shell $next_shell"
          status=$shell_status
          echo "Slave $1: ERROR IN Shell $next_shell"
          status=$shell_status >> $errfile
        fi #
        # message "Slave $1: Finished Shell $next_shell"
        # message
        done # all done
      message "Slave $1: Done (Executed $shell_count Shells)"
        # message
      return # slave complete
    done # all done
}
# paresh_driver
# This code is executed by the top level process only. It
# parses the arguments and spawns the appropriate number
# of slaves. Note that the slaves run this same shell file,
# but the slaves execute different code, based on the
# exported variable PARESH.
#-------------------------------------------------------------
# paresh_driver()
{
  rm -f $lklogfile # start a new log file
  if [ "$1" = "" ] # first argument?
    then # no?
      master_file="master.list" # default value
    else # yes?
      if [ ! -f "$1" ] # does file exist?
        then # no?
          echo "$0: Unable to find File $1"
            # say so
        exit 1 # quit
      else # yes?
        master_file="$1" # use specified filename
      fi
    fi
  if [ "$2" = "" ] # Second Argument?
    then # no?
      parallel_count=4# default value
    else # Yes?
      if [ "$2" -lt 1 ] # Less than 1?
        then # Yes?
          echo "$0: Parallel Process Count Must be > 0"
            # message
        exit 1 # quit
      else # no?
        parallel_count=$2 # Use Specified Count
      fi
    fi
message "-----------------------------------"  # Startup Banner
message "Master Process ID: $PARESH"
message "Processing File: $master_file"
message "Parallel Count: $parallel_count"
message "Log File: $logfile"
message "-----------------------------------"

cp $master_file $workfile  # make a copy of commands file
while test $parallel_count -gt 0  # Have we started all slaves?
do  # Not yet
    if [ ! -s $workfile ]  # Is there work to do?
        then  # No?
            message "All Work Completed - Stopped Spawning at $parallel_count"
            break  # Quit spawning
        fi
    $0 $parallel_count  # spawn a slave (with slave #)
    message "Spawned Slave $parallel_count [pid $!]
    parallel_count=`expr $parallel_count - 1`  # decrement counter
done  # Next
wait  # Wait for all slaves
message "All Done"  # message
return  # Function Complete
}
#-------------------------------------------------------------
# main
# This is the main section of the program. Because this shell
# file calls itself, it uses a variable to establish whether or
# not it is in Driver Mode or Slave Mode.
#-------------------------------------------------------------
#
if [ "$PARESH" != "" ]# If variable is set
then # then slave mode
    workfile=/tmp/paresh.work.$PARESH # Work file with parent pid
    lockfile=/tmp/paresh.lock.$PARESH # Lock file with parent pid
    lklogfile=/tmp/paresh.lklog.$PARESH
    # LockLog file with
    # parent pid
    logfile=/tmp/paresh.log.$PARESH # Log File with parent pid
    errfile=/tmp/paresh.err.$PARESH # Error File with parent pid
    paresh_slave $*
    # Execute Slave Code
else
    PARESH="$$"; export PARESH # Establish Parent pid
    workfile=/tmp/paresh.work.$PARESH # Work File with parent pid
    lockfile=/tmp/paresh.lock.$PARESH # Lock File with parent pid
    lklogfile=/tmp/paresh.lklog.$PARESH
    # LockLog File with
    # parent pid
    logfile=/tmp/paresh.log.$PARESH # Log File with parent pid
    errfile=/tmp/paresh.err.$PARESH # Error File with parent pid
    rm -f $errfile # remove error file
    paresh_driver $*
    # execute Driver Code
    rm -f $workfile # remove work file
    rm -f $lklogfile # remove lock log file
    if [ -f $errfile ] # Is there was an error
        then
            message "********************************************************************************
            message "FINAL ERROR SUMMARY. Errors logged in $errfile"
cat $errfile | tee -a $logfile
            message "********************************************************************************
            exit 1
        fi
fi
exit
Conclusion

In this article, a script is described that performs considerably fast Oracle database exports by using parallel processing on SMP machines. This script has been proven invaluable for situations where it is needed to perform exports of large mission-critical databases that must be highly available.

The script was contributed by its authors, Neal Sundberg and Mike Ellison of QUALCOMM Incorporated.

Author’s Bio:

Stan Stringfellow is an independent author, technical writer and software developer who is currently doing work for the Sun BluePrints program. He holds a B.A.C.S. from UC San Diego and has written many manuals including the original software manuals for the Sun Enterprise 10000 (Starfire™). He may be contacted at stan@stringfellow.com.