DBMS_SCHEDULER NOT JUST A REVAMP OF DBMS_JOB

Rune Mørk, Tia technology

Dbms_scheduler is put to market by Oracle as the new dbms_job, and indeed it is, but it is also much much more.

With dbms_scheduler you can control your programs and your scheduler mechanisms in a richer environment, enabling users to control resource allocated to jobs, ordering the execution of a job or groups of jobs, have jobs switching from one resource plan to another during the day and much more.

My aim with this article is to show how the scheduler is organized and used and illustrate best practices.

Database tables influencing dbms_scheduler

Dbms_scheduler is utilizing a number of database tables; these can be difficult to get an overview of regarding their interaction, due to Oracles poor documentation on this. If you consult figure 1, I have shown those tables (view's) who has impact of how dbms_scheduler can work.

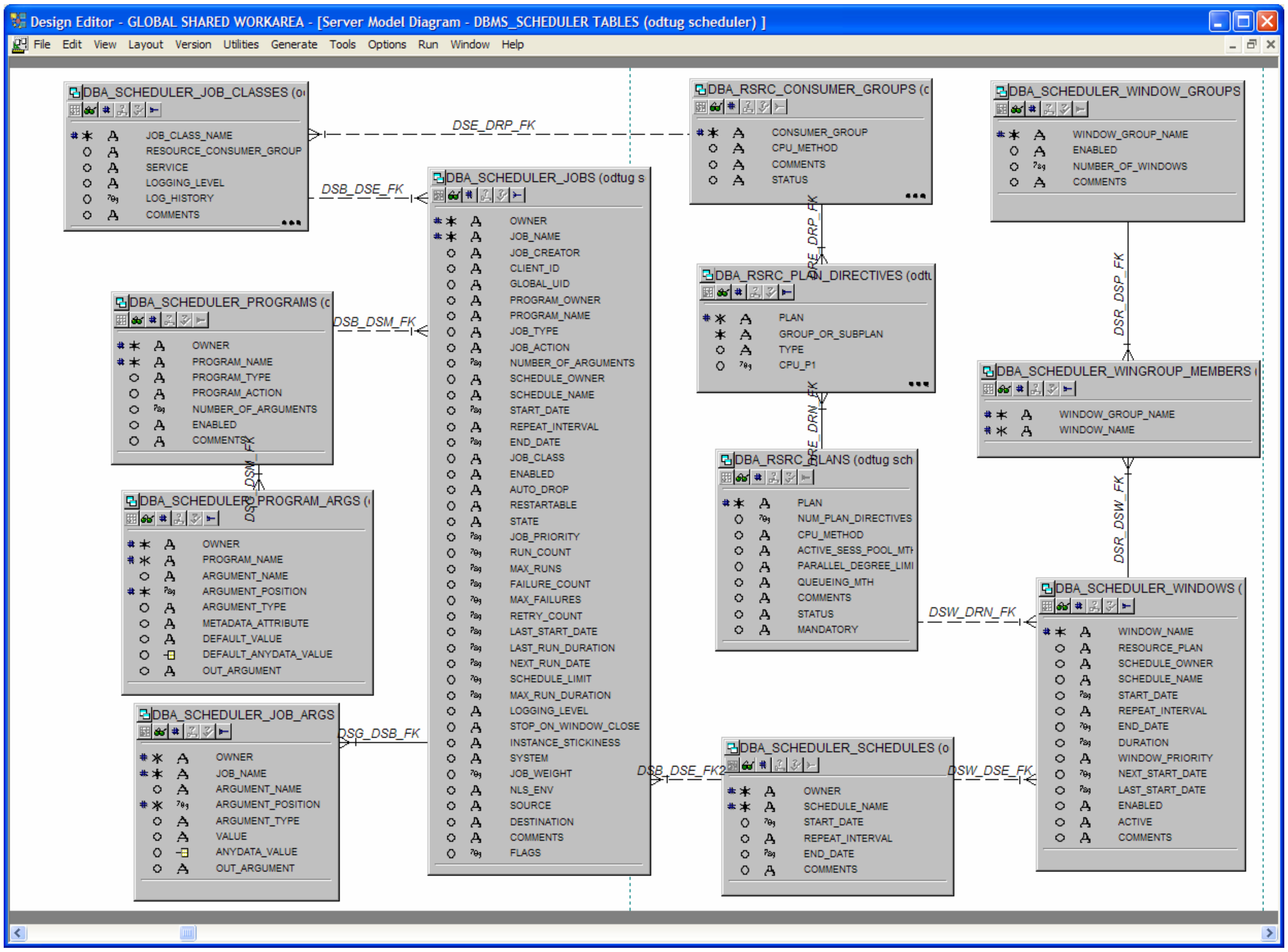

Figure 1 Data diagram showing views in dbms_scheduler

Trying to understand the impact of figure 1 and how the different objects interact can be quite difficult. I have taken the approach to explain how to use dbms_scheduler bottom up, staring with the program/job, then schedules, job classes, windows, window groups, and ending with a best practice chapter. In between I had to put a small chapter regarding plan and resources, because this feature is used by windows and job classes.

Sample program

Throughout this article I will be using a sample program listed below

```
create or replace procedure sample_program(
       p_sec in number,
       p_name in varchar2,
       p_loop in number default 0) is
begin
  insert into sample_table
  values(to_char(sysdate,'ddmmyyyy hh24:mi:ss')|| ' program '||p_name ||' start ');
  dbms_lock.sleep(p_sec);
  for j in 1..p_loop loop
    for i in 1..1000000 loop
      null;
    end loop;
  end loop;
  insert into sample_table
  values(to_char(sysdate,'ddmmyyyy hh24:mi:ss')|| ' program '||p_name||' end ');
  commit;
end;
```

This program is not very smart, but for the illustration ….

Objects in DBMS_SCHEDULER

Dbms_scheduler contains a lot of packages and procedures that to some extent is explained in the manuals. My aim here is to mention those that you need to know in order to create a working environment for batch scheduling.

Running a simple job or what you could do in dbms_job

When being a Oracle Old Timer as I am, running a program with dbms_job is quite easy, dbms_scheduler offers the same functionality, just tweaked a little.

To run a simple program you simply type :

```
begin
  dbms_scheduler.create_job(job_name  => 'test_job1',
                            job_type  => 'plsql_block',
                            job_action=> 'sample_program(5, ''program 2'',1000);'
  );
  dbms_scheduler.enable('TEST_JOB1');
  commit;
end;
```

Before and during the run you can monitor the job by issuing

```
select * from USER_SCHEDULER_JOBS
```

where you can see the jobs currently scheduled, and running. If the job is submitted as above, just running one, it will only appear in this list until the run is completed.

You can off course also set the job up to be run at a later time but that feature I will save till later when we have all the concept in play.

Programs

Often you would not just use the create job functionality, that is only the case of you are in a situation where you are executing a one off run, in most cases you would like to set programs up to run again and again, this is where programs come into play, and dbms_scheduler start to be more fun.

A program is a metadata description of database object of witch you can order an execution.

In order to set up a program you must do the following;

```
begin
  dbms_scheduler.drop_program('test_prog1');
  dbms_scheduler.create_program(program_name => 'test_prog1',
                                program_type => 'stored_procedure',
                                program_action=> 'sample_program',
                                number_of_arguments => 3);

  dbms_scheduler.define_program_argument(
                                program_name => 'test_prog1',
                                argument_position => 1,
                                argument_name => 'p_sec',
                                argument_type => 'NUMBER');

  dbms_scheduler.define_program_argument(
                                program_name => 'test_prog1',
                                argument_position => 2,
                                argument_name => 'p_name',
                                argument_type => 'VARCHAR2');

  dbms_scheduler.define_program_argument(
                                program_name => 'test_prog1',
                                argument_position => 3,
                                argument_name => 'p_loop',
                                argument_type => 'NUMBER');
  dbms_scheduler.enable('test_prog1');
  commit;
end;
```

Notice that I this time use the program_type => 'stored_procedure', if I was using plsql_block as before I'am unable to define arguments to this procedure, and that would not be desirable, this feature is undocumented, so take my word for it.

Also note that I use the enable procedure of the package, I could have used the enable parameter of the create job package, but as you can see, the enable parameter is by default false, so you either need to enable the program manually or use the enable parameter.

Having this program in place I can now initiate the job by doing the following:

```
begin
  dbms_scheduler.create_job(job_name => 'test_job1',
                            program_name => 'test_prog1');

  dbms_scheduler.set_job_argument_value(job_name => 'test_job1',
                                        argument_position => 1,
                                        argument_value => 5);

  dbms_scheduler.set_job_argument_value(job_name => 'test_job1',
                                        argument_position => 2,
                                             argument_value => 'program 3');

     dbms_scheduler.set_job_argument_value(job_name => 'test_job1',
                                             argument_position => 3,
                                             argument_value => 1000);

     dbms_scheduler.enable('TEST_JOB1');
     commit;
   end;
```

So with this in place I can schedule a program at a interval of my choice, this could be done by using the arguments start date, and repeat interval. This is nice to know but you should instead focus on the schedules and use those.

Schedule

A schedule should be created to you have control of when your jobs is running, by default some schedules are already in place, but you should create your own, like in the following examples:

```
   BEGIN
      dbms_scheduler.create_schedule(
          schedule_name   => 'EndOfMonth'
         ,repeat_interval => 'FREQ=Monthly;BYMONTHDAY=-1'
         ,comments        => 'Standard End-Of-Month Schedule');
   End;
```

This will create a end of month schedule, but at what hour?

Understanding how the repeat interval is build up is important, you can consult the documentation for a full picture, but here are a few examples:

Frequency	When
'FREQ=Minutely;Interval=1'	Every minute
‘FREQ=Daily;Byhour=6’	Every day at 6 am + the minutes when you applied it
‘FREQ=Daily;Byhour=6;byminute=33’	Every day at 6.33 am + the seconds when you allied it
‘FREQ=Monthly;byday=-1SUN;byhour=21;byminute=00;	Last Sunday of the month at 9 pm.

Note that if a frequency is created without enough precision, then the creation time for the job is determining the rest of the precision.

If you are not sure how your repeat interval will work I’ve created a small program you can use, this will return how often your job will be run with the current settings:

```
   CREATE OR REPLACE PROCEDURE show_job_timing(
        p_Schedulename IN VARCHAR2
       ,p_Iterations     IN NUMBER)
   AS
       -- Local variables
       w_begin       TIMESTAMP;
       w_next        TIMESTAMP;
       w_returned    TIMESTAMP;
       w_calendar_string varchar2(300);
       cursor c_ss is
       select repeat_interval, TO_TIMESTAMP_TZ(start_date)
```

```
      into w_calendar_string, w_begin
      from DBA_scheduler_schedules
      where schedule_name = p_schedulename;
BEGIN
      -- Set the returned date to the start date (for 1st iteration)
      open c_ss;
      fetch c_ss into w_calendar_string, w_begin;
      if c_ss%notfound then
        DBMS_OUTPUT.PUT_LINE('Schedule not found');
      else
        DBMS_OUTPUT.PUT_LINE('>>>');
        DBMS_OUTPUT.PUT_LINE(
          'Next ' || p_Iterations ||
          ' iterations of Schedule ' || p_Schedulename ||
          ' as of ' || w_begin);
        for idx IN 1..p_Iterations
          loop
              -- Evaluate the calendar string
              DBMS_SCHEDULER.EVALUATE_CALENDAR_STRING(
                   w_calendar_string
                  ,w_begin
                  ,w_returned
                  ,w_next);
              -- Print the output
              DBMS_OUTPUT.PUT_LINE(
                  'Iteration #' || TO_CHAR(idx, '99999') ||
                  ' will run on  ' || w_next
               );
              -- Set up the next iteration
              w_returned := w_next;
        end loop;
      end if;
      close c_ss;
END;
```

Then you can use the schedule for the job like:

```
begin
  dbms_scheduler.set_attribute('test_job1',
                               'schedule_name',
                               'everyminute');
end;
```

Now if for any reason your job is not running, you can look into the job log to see why, that is possible with the following query:

```
select * from dba_scheduler_job_run_details
order by actual_start_date desc
```

There can be a number of reasons why your schedules job appear not to be running, it could be that the program that the job uses has been disabled for some reason, then you will be able to see this in the run_details view.

When using schedules do bear in mind that if your application is spanning more timezones you might think about setting the scheduler attribute, default time zone, so all start dates will be calculated according to that.

Job class

If you are working in a RAC environment you should utilize job classes in order to distribute your workload on difference services, ie on different database instances. This feature can be used if you would like to control and distribute your workload manually to the database servers.

You can also use the job classes to control the usage of resources via the resource consumer group parameter, you can use when creating the class. The use of consumer groups and services are mutually exclusive, so it’s either one or the other.

If you are not interested in the above 2 usages of job classes, there is one last reason why you should use them, logging. With the parameters logging level and log history you can control the amount of logging you want for all the jobs belonging to a specific class. Logging level can be set as listed in the tables below:

Value	Meaning
Dbms_scheduler.logging_off	No logging for any job in this class
Dbms_scheduler.logging_runs	Only log starts of a run for any job in this class
Dbms_scheduler.logging_full	As above but also keep information of any tampering with any job in this class, (creation, enable, disable, alter etc), ie an audit trail for the settings of jobs.

Log history can be set to the number of days you would like to keep the log, default is 30 days, this can be changed on scheduler level but then it pertain all job classes.

To create a job class and use it you can do the following:

```
begin
  dbms_scheduler.create_job_class(job_class_name          => 'class_full_log',
                                  resource_consumer_group => NULL,
                                  service                 => NULL,
                                  logging_level           => DBMS_SCHEDULER.LOGGING_FULL,
                                  log_history             => 100,
                                  comments                => 'full logging 100 days');
end;
```

Now the job class is ready to be used, apply it to the job we have created:

```
begin
  dbms_scheduler.set_attribute('test_job1',
                               'job_class',
                               'class_full_log') ;
end;
```

then you can follow what is happening to your jobs by issuing the following:

```
select * from dba_scheduler_job_run_details order by log_date desc;
```

and you can follow who is altering jobs in this class by:

```
select * from DBA_SCHEDULER_JOB_LOG
```

Notice that you can add the `where operation <> ‘RUN’` to the statement above and you will get the incidents where someone tried to alter your jobs.

Window

The scheduler environment offers the opportunity to group job together in with the window concept.

The window enables you to specify an inline schedule or use a predefined schedule; I would urge you to use the last option. Window can, and should, also be used to control the use or resources, by referencing to a predefined resource plan. In my example I've created a resource plan the following way:

```
begin
  DBMS_RESOURCE_MANAGER.CREATE_PENDING_AREA;
  DBMS_RESOURCE_MANAGER.CREATE_PLAN(
   PLAN    => 'my_resource_plan',
   CPU_MTH => 'EMPHASIS',
   comment => 'test spilt plan');

   DBMS_RESOURCE_MANAGER.CREATE_CONSUMER_GROUP
   (CONSUMER_GROUP => 'cpu1',
    COMMENT        => 'cpu1');

    DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE
   (PLAN             => 'my_resource_plan',
    GROUP_OR_SUBPLAN => 'cpu1',
    COMMENT          => '30% on cpu1',
    CPU_P1           => 30);

    DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE (
     PLAN             => 'my_resource_plan',
     GROUP_OR_SUBPLAN => 'OTHER_GROUPS',
     COMMENT          => 'this one is required',
     CPU_P1           => 0,
     CPU_P2           => 100);

    DBMS_RESOURCE_MANAGER.SUBMIT_PENDING_AREA;
end;
```

So to create a schedule for the window, do the following:

```
BEGIN
   DBMS_SCHEDULER.CREATE_SCHEDULE(
   schedule_name   => 'Everyday'
  ,repeat_interval => 'FREQ=daily; BYHOUR=9'
  ,comments        => 'test2'
  ,start_date      => trunc(sysdate) + 8/24);
End;
```

And to create a window:

```
begin
  dbms_scheduler.create_window(window_name    => 'test_window1',
                               resource_plan  =>  'my_resource_plan',
                               schedule_name  => 'TIA.EVERYday',
                               duration       => INTERVAL '5' HOUR
                               );
end;
```

Now this window opens every day at 9 am and lasts for 5 hours, every job executed in this window will use my_resource_plan for execution.

You can open and close the window to control this fact by the following code,

```
begin
  dbms_scheduler.CLOSE_window(upper('sys.test_window1'));
```

```
end;

begin
  dbms_scheduler.OPEN_window(upper('sys.test_window1'), interval '5' hour);
end;
```

and then create a job to control if resource plans has been changed.

This job can select the current plan from v$rsrc_plan and the current consumer group from v$session.

To get all this to work you must remember to enable a default user defined plan, like in the following.

```
ALTER SYSTEM SET RESOURCE_MANAGER_PLAN='MY_RESOURCE_PLAN222';
```

If you omit this part, the job will continue on the old plan not using the plan defined with your window, and that is a pity, I have not been able to find a way around this, so you should have 2 plans.

When using your windows you can also monitor when windows open and close by using the following view:

```
select * from dba_scheduler_window_details
```

When investigating for this article, I came across a interesting parameter, stop_on_window_close that could be set as an attribute on a job. You cannot set it when creating the job but apply it as a set attribute thing. Despite what associations you might get on the name of the parameter, it does what is says, if the job is running, stop on window close is set to true, and the window close, the job stops, but only until the next time it is scheduled, then it starts again. So this parameter is, in my opinion there only to help jobs switching resource plans. If you want a job permanently stopped, you should not only close the window, but also disable the job itself.

Window groups

On top of the windows you can group these together in window groups in order to control them in a bundle. With windows grouped together you can enable and disable the windows as a whole, besides this you have no other use for window groups. I expect in the future other possibilities will emerge for window groups.

To create a window group you do the following:

```
Begin
  dbms_scheduler.create_window_group(
     Group_name  => 'Window_grp1',
     Window_list => 'test_window1','test_vindow2',
     Comments    => 'test comment');
End;
```

You can always add new window group members with dbms_scheduler.add_window_group_member, as well as removing the member with remove_window_group_member.

With the group in place, you can enable or disable the window group by …

```
begin
  dbms_scheduler.enable('sys.Window_grp1');
end;
```

or

```
begin
  dbms_scheduler.disable('sys.Window_grp1');
end;
```

Best practice
Now the big question that remains is what of all these offerings should we utilize, and witch should we bypass.

You could go for every usable feature available or you could just bypass the framework and revert just to use dbms_scheduler as dbms_job, it all depends on your system and the requirements.

My company is offering a framework solution in insurance management, and our solution should fit both small and large business, so therefore we had to find the golden solution that would, more or less, fit anyone. To do so we are creating our own staging tables with scheduler information, and then creating the scheduler mechanisms based on those.

We do offer a default solution for our entire batch processing, and some of this I've included here in the best practice chapter.

Programs
You should describe your batch related programs in a structured way, so you should create them as programs in the scheduler environment.

Job classes
You should create (at least) one job class to associate with your jobs when submitting them, this job class should be associated with a resource consumer group(batch), allowing for resource management. Depending on your auditing needs consider the logging level.

Schedule
Create all the schedules you anticipate to use, if you have the schedules in place, it is easier to see how they relate to each other.

Job
Initiate your job using a schedule, and a job class.

Windows
You should have 4 windows each with relation to a resource plans as shown in this table:

Window	**When**	**Start**	**End**	**Resource plan**
Weekday Night	Mon-fri	20.00	05.00	Night
Weekend Night	Sat-sun	16.00	05.00	Night
Weekday Day	Mon-fri	05.00	20.00	Day
Weekend Day	Sat-sun	05.00	16.00	Day

These windows are connected to resource plans Night and Day, and they are defined as follows:

Resource Plan	**Consumer group**	**Cpu usage**
Night	Batch	100
Day	Batch	10

With this scheme, we are able to allow batch running during the day, and the batch will not (significantly) impact the on-line part of the system.

You should note here that that setting the cpu usage to 10%, does not necessarily mean that the usage is restricted to 10%, the resource manager can choose to allow more that 10% cpu usage, if nothing else is happening on the server.

You could also choose only to have the day window in order to restrict the day batch part to only use cpu limited, I've chosen to have both day and night resource plan with the possibility later to divide jobs into 3 groupings, so you can say that is for future use.

Window_groups

So far I can't see much use for window groups, you could bundle your weekday into one group and your night into another etc, but in a limited setup as this I do not see much benefit with this step.

Conclusion

Going back to the figure showing all the different entities available to dbms_scheduler, you can see that a whole new world is opening for us.

I would urge you to investigate all the possibilities in this package, and use them, ie to allow batch to run during the day, and structuring your execution.

About the author

Rune graduated in 1990 as M. Sc. in it and operational analysis from Technical University of Copenhagen. Since then Rune has worked for a number of consultancy companies in various sizes, from his own company to Oracle Consulting. Rune has presented at ODTUG 2001, 2002, 2005, 2006 and 2007, Danish oracle ekspert conference 2003, 2004 and 2005, and ioug collaboration 2006 and 2007.