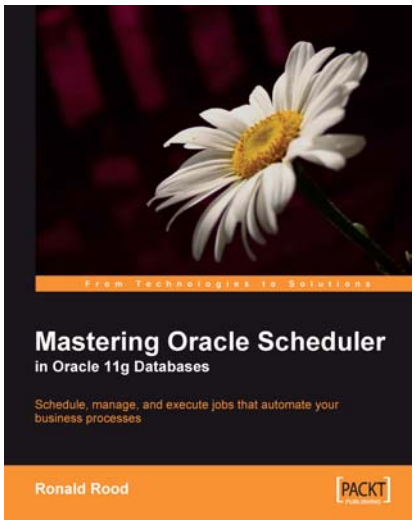
Mastering Oracle Scheduler in Oracle 11g Databases

Ronald Rood



Chapter No. 3
"Control the Scheduler"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "Control the Scheduler"

A synopsis of the book's content

Information on where to buy this book

About the Author

Ronald Rood is an IT professional for over 20 years. His private domain is named after the first account that was created on a computer for him. That account was *ronr* on a DPS9 system, and so his private domain is called `ronr.nl` where you can contact him using `ronr@ronr.nl`. He eagerly joined the Oracle community and became a skilled, innovating DBA and troubleshooter. He is a master of C, PRO*C, lots of scripting languages, and—of course—PL/SQL. Ronald's real power is in the combination of the rich Unix world and Oracle. According to him, there is no such thing as a problem that cannot be solved; it just might take a little time.

Ronald speaks several languages such as Dutch, English, German, and some French. In the private time that he shares with his wife and two children, he likes to take an occasional dive (from the sky), fl y with radio-controlled models, ride recumbents, and work as a volunteer for a local water scouts group.

Currently, Ronald is one of the Ciber Oracle consultants in The Netherlands and cooperates in many projects for many large companies. Before writing this book, Ronald wrote Oracle-related articles for the Dutch Oracle user group magazine. On his blog, you can find some short articles about various items, but most are related to Oracle.

I would like to thank everyone who helped me writing this book. Not in the least, Sandra for having lots of patience for me while writing, Silvana and Alex for being great children I can be proud of, Robin and the other reviewers for challenging me to write better, and Harold for showing me that there is a way that might or might not be seen. I also would like to thank my colleagues of Ciber for being my colleagues, and Mark for giving me the challenges that I like so much. Without my parents, I would never have existed at all, but they gave me the opportunity to study without ever asking anything in return. Bedankt ma, bedankt pa zonder jullie was het nooit wat geworden.

Mastering Oracle Scheduler in Oracle 11g Databases

Welcome to the world of Oracle Scheduler! Oracle Scheduler is a free utility included in the Oracle database that makes the Oracle RDBMS the most powerful scheduling tool on our planet (and in the known parts of the galaxy).

An overview of Oracle Scheduler

The Oracle Scheduler can be used to automate not only the simple maintenance tasks, but also the complex business logic. Traditionally, only PL/SQL could be executed in the Scheduler. Later, operating system scripts were added to it, and now we can run jobs on remote systems and cross platform as well. This can turn the Oracle Scheduler into the spider in your Web, controlling all the jobs running in the organization and giving you a single point for control.

Database background

Relational database management systems (RDBMS) can be very powerful. With a little code, we can use the RDBMS as a filesystem, mail server, HTTP server, and now also as a full-blown job Scheduler that can compete very well with other commercial job Schedulers. The advantage that Oracle gives us is in terms of price, flexibility, and phenomenal power. The scheduling capabilities are all a part of the normal Oracle license for the RDBMS, whereas others have a serious price tag and often require a database for the repository to store the metadata of the jobs.

Scheduling events in the database

Since Oracle added the procedural option to the database, they also included some scheduling power provided by `dbms_jobs`. Although a bit restricted, it was used extensively. However, no one would even think about using this as an Enterprise-level Scheduler tool. This changed when Oracle introduced 10gR2. In this release, Oracle could not only start jobs that ran outside the database, but they also added the job chain.

In 11g, Oracle also added the option to run jobs on remote systems where no database is running. Now it's time to rethink what the database actually is. In the early days, a database was just a bunch of code that could hold data in tables. Today, the Oracle RDBMS can do that—and that too well—along with many more things. In fact, the Oracle RDBMS can perform so many tasks so amazingly, that it's surprising that we still call it just a database. We could easily turn it into a personal assistant.

Oracle Scheduler 11g can:

- Run jobs where no database ever was before
- Use different operating system credentials per job
- React on events
- Schedule jobs on multiple platforms simultaneously
- Give a tight security

What This Book Covers

Chapter 1 will get you going with the Scheduler as quickly as possible. In the end, you will automate simple tasks that are now maintained in cron, task manager, or the good old **DBMS_JOB** package, for example.

Chapter 2 will show you a lot of possibilities of chains with many examples and explanations. In short, it will tell you all you ever wanted to know about chains, but were afraid to ask.

Chapter 3 is for all you people living in an organization that requires strict job separation. This chapter will show how to make good use of the Scheduler and apply job separation.

Chapter 4 is a very important chapter that explains how to crank up the power of a system to the limits by combining the Scheduler and the Resource Manager. Here you will find how to get the best out of your system.

Chapter 5 will be of a great help in setting up remote external jobs introduced in Oracle 11g. How is this related to the old-fashioned local external jobs that we know since Oracle 10g and why we should get rid of the old external jobs? Get your answers here.

Chapter 6 helps the reader to get a firm grip on events and explains how to make good use of events. Events sound like voodoo, but in the end are an extra tool found in the Scheduler.

Chapter 7 considers the fact that when the jobs get more complicated, it gets harder to understand why something works differently than planned. This chapter gives the reader a fresh look at how to follow and debug Scheduler jobs.

Chapter 8 will give you some creative implementations of more or less common tasks—this time implemented using the Scheduler. This chapter gives a working code with clear explanations. This broadens the horizon and will take down the barriers that might exist between other environments and Oracle.

Chapter 9 shows how the Scheduler can be used in other configurations such as standby databases and RAC.

Chapter 10 shows how the Scheduler can be managed and monitored remotely through a web interface.

3
Control the Scheduler

Oracle Scheduler does a lot of out of the box things, and if the demands are not too high, Oracle Scheduler can cover most of the situations. When your application has to run many thousands of jobs in an hour and log them over a long period, a little more attention is needed. In cases where the demands are high, we need to take care of a few more things to keep the system happy. What we need to do depends on *how* the Scheduler is used and *what* kind of load it is supposed to handle.

In this chapter, we will take a closer look at how we can control this beast. We will take a look at the privileges for job creation, job execution, and Scheduler management. We will also examine how to control logging retention and find a way to prevent jobs from running when the database starts.

Job creation

Like most object types in the database, as seen at the beginning of Chapter 1, Oracle enables us to create Scheduler objects. The privileges `create job` and `create external job` are very important. They should normally be used when building an application system. Also, there is a `create any job` privilege, which can be useful when you need to create a job in a different schema. Normally, this privilege should not be granted to anyone. It will allow the grantee to run an arbitrary code on any schema, which is not particularly desirable. Instead, just log on to the correct schema and perform the tasks using the correct privileges. The following privileges can be used:

System Privilege	Description
`create jobs`	This privilege allows the grantee to create not only jobs but also chains, schedules, and programs. A schema can always alter and drop the scheduler objects in their own schemas even without the `create jobs` privilege. For chains, we need more privileges.
`create external jobs`	Using this privilege, we can create external jobs. Note that when the job references to a program that uses an executable, this privilege is also needed. In case another user created an external job in a schema, that schema still needs `create external jobs` to run that job.
`create any jobs`	This privilege is very powerful and should preferably not be granted to anyone. It allows a user to create a job in any schema, except `SYS`. The end effect is that the grantee is able to run arbitrary code in any schema, which should not be allowed.
`execute any class`	Using this privilege, we can have a job running in every job class that we want.
`execute any program`	Allows the grantee to use any program in a job.
`manage scheduler`	This privilege gives the possibility to manage job classes, windows, window groups, and logs. It also allows the grantee to set the Scheduler attributes.
`dbms_rule_adm.grant_system_privilege (dbms_rule_adm.create_rule_obj, '<schema_name>')`	Needed to be able to create chains.
`dbms_rule_adm.grant_system_privilege (dbms_rule_adm.create_rule_set_obj, '<schema_name>')`	Needed to be able to create chains.
`dbms_rule_adm.grant_system_privilege (dbms_rule_adm.create_evaluation_ context_obj,'<schema_name>')`	Needed to be able to create chains.

Here, it is interesting to note that the `drop any job` privilege seems to be missing. For other object types, we have the `create any` privilege and also the `drop any` privilege. Making use of these *any* privileges looks smart at first glance. However, it makes a system less transparent and more difficult to maintain eventually. Preferably, objects are granted explicitly instead of falling back on *any* privileges when building an application. For example, select any table and you will observe that it is much harder to find how an application flows or what the impact of dropping an object is. If the privileges are explicitly granted, we can see that someone is using our object.

Mostly, it is smarter to create schemas using the least privileges principle. This means that one or more schemas contain tables that hold the data, and other schemas contain the procedures that act on the various tables. Users or roles should have privileges on the procedures.

The `DBMS_SCHEDULER` package is available to the public. The use of the package is controlled by the underlying privileges such as `create jobs`, `manage scheduler`, and the execution privileges on the products of the `DBMS_SCHEDULER` package such as jobs, programs, and job classes.

Privileges	Object type	Description
`execute`	`job_class`	To be able to use a job class, we need `execute any job` or `execute any class`, or `execute` privilege on the specified job class. Job classes are in the `SYS` schema. So when granting them, we need to prefix the job class with `sys`.
`execute`	`Program`	When we cannot use the `execute any program` privilege. We need an `execute` privilege on the specified program if it is in another schema.
`execute`	`Credential`	Needed to be able to use the credential if it is in a different schema.
`execute`	`Job`	Needed to be able to run a job from another schema.

Now we have enough information to create a job as far as the job system is concerned. The application-specific privileges that enable the job to perform its task should hopefully also be granted as a matter of course. Don't forget the synonyms. Having privileges is fine, but if the objects are not visible to you because you missed the public or private synonyms on the procedure, the job will fail. To avoid this failure, you can prefix the objects with the owner name or change the current schema using this:

```
alter session set current_schema;
```

Whether or not prefixing should be preferred over using synonyms is not the point of discussion of this book.

During job creation or alteration, we can assign a job to a job class. The default job class is `DEFAULT_JOB_CLASS`. The job class can be used to assign a job to a resource consumer group. It also specifies the logging level, the log history, and the service name that should be used for the class.

The `job_class` defines the resource consumer group that the job has to use. The logging level and the log retention are also defined by the job class. For the log settings, the job class gives more granularity than falling back to the Scheduler attributes without having to define the logging properties for each and every job. In addition to that, the job class is also the interface for service selection. Oracle is mainly aiming at having multiple applications in a database where each application can have a service name defined for itself. In doing so, we can choose the instance where the specific service is running. A service can run in multiple instances.

The logging level can be one of the following:

- `DBMS_SCHEDULER.LOGGING_OFF`: No logging at all.
- `DBMS_SCHEDULER.LOGGING_RUNS`: The starts and stops are recorded with timestamps and status information.
- `DBMS_SCHEDULER.FAILED_RUNS`: This logs only the failed runs.
- `DBMS_SCHEDULER.LOGGING_FULL`: This records not only the runs, but also the job creations and alterations. This enables us to see when a job was created, changed, enabled, disabled, or dropped. When you suffer from a lot of unexplained variations during the runtime, it can be useful to set the logging level to `FULL`. When the job is dropped, the fact that it was dropped as well as the definition of the job are logged.

Log history specifies the number of days the Scheduler should retain the log entries. Log entries for a chain are purged only when the chain has ended. This might be the cause of flooding in the log tables when you fix a problem that causes a chain to stall. However, you might have to forget to make the chain end. The log history is specified in days. It would be very useful to be able to specify the number of job runs to be retained in the logging. In order to be able to compare the last five runs of an end-of-year run, we are now forced to retain the logging of five years. The valid range of values is `NULL`, `0` (no logging at all) to `999`. When `NULL` is specified, the log history is inherited from the global Scheduler attribute.

As the logging of Scheduler jobs can grow quickly, it is good to know which tables are under the Scheduler log views. There are two tables that contain the job logs—the `SCHEDULER$_EVENT_LOG` table, (which also holds the Windows log) and the `SCHEDULER$_JOB_RUN_DETAILS` table. These tables are located in the `SYSAUX` tablespace. According to the `v$SYSAUX_OCCUPANTS` view, we cannot move them to another tablespace. This can be seen using the following query:

```
SELECT occupant_name, occupant_desc, move_procedure_desc,
space_usage_kbytes
FROM V_$SYSAUX_OCCUPANTS
WHERE occupant_name = 'JOB_SCHEDULER';
```

This gives us the following output:

```
OCCUPANT_NAME  OCCUPANT_DESC  MOVE_PROCEDURE_DESC  SPACE_USAGE_KBYTES
JOB_SCHEDULER  Unified Job    *** MOVE PROCEDURE   50368
               Scheduler      NOT APPLICABLE ***
```

So we need to monitor the `sysaux tablespace` usage closely when we use large volumes of jobs that use logging.

The service name that is specified in the `job_class` (which we connect the job to) is useful for tying a job to a specific service name in an RAC configuration. This can be used when there is a specially configured instance available in an RAC database that runs jobs, while other instances serve online users.

In the `ALL_SCHEDULER_JOB_CLASSES` view, we can see which classes exist and how they are defined. There are quite a few classes defined in an empty database. Oracle uses them for many of the automated background tasks such as statistics gathering, log purging, auto space, and advisory.

Other attributes of a job, which might be handy, are `job_priority`, `schedule_limit`, `restartable`, and `max_run_duration`. Each of these attributes can only be set using `DBMS_SCHEDULER SET_ATTRIBUTE`. Let's look at each of these attributes here:

- `job_priority`: This gives the Scheduler the power to select a higher priority job before a lower priority job. The priority count ranges from `1` to `5`, where the highest priority is `1`, the lowest is `5`, and the default is `3`. An example where using a job priority can be useful is the situation where we generate a statistics collection job for every object (table, table partition, or index) that has stale statistics or no statistics at all. All the jobs can be generated using the default priority and have the same job class. But due to the enormous impact of not having statistics at all, we should give highest priority to the jobs that generate statistics for an object that has no statistics to make sure that those statistics are generated the fastest. The use of `job_priority` only makes sense when the jobs that differ in priority are in the same job class and have the same `start_date`. Here is an example:

```
dbms_scheduler.set_attribute
  (
    name      => l_job_name,
    attribute => 'start_date',
    value     => NULL
  );
if i.last_analyzed is null
then
-- top prio for objects without stats !
dbms_scheduler.set_attribute
  (
    name      => l_job_name,
    attribute => 'job_priority',
    value     => 1
  );
end if;
```

- `schedule_limit`: This is meant to help with the decision to run a job later than the scheduled time (if the system is very busy) or to reschedule the job to a next scheduled time. If a specific task has to be completed before 09:00, it does not make sense to start it at 08:00 when normally, the task takes 4 hours to complete. The schedule limit is specified in minutes from the scheduled time. This parameter only makes sense for a repetitive job. Here is an example:

```
BEGIN
  sys.dbms_scheduler.create_job
   (
    job_name       => 'test',
    job_type       => 'PLSQL_BLOCK',
    job_action     => 'begin
 -- Insert PL/SQL code here
                        end;',
    repeat_interval => 'FREQ=DAILY;BYHOUR=4;BYMINUTE=10',
 -- should start at 04:10 (not guaranteed to start at this time)
    start_date      => systimestamp at time zone 'Europe/Amsterdam',
 -- available for scheduling immediatly
    job_class      => '"DEFAULT_JOB_CLASS"',
    comments       => 'testjob',
    auto_drop      => FALSE,
    enabled        => FALSE
   );
  sys.dbms_scheduler.set_attribute
   (
    name           => 'TEST',
    attribute      => 'schedule_limit',
    value          => numtodsinterval(240, 'minute')
   );
```

```
-- it does not make sense to start this job after 08:10
-- if the job is not started before 08:10, forget this run and
-- use the next schedule time (tomorrow at 04:10)
   sys.dbms_scheduler.enable( 'TEST' );
END;
```

- `max_run_duration`: This can help in making the decision about whether to stop or continue the job after it exceeds the maximum run duration. In such a case, the `job_over_max_dur` event is raised. (The job is not automatically stopped.) This is a more advanced parameter requiring a job event handler process that reads the event from the job event queue and notifies a user using mail, SMS, or whatever is appropriate. In a real-life example scenario, this is used to generate a notification when a back-up job takes more time than usually expected.
- `restartable`: This can be used to make the job restart if an error occurs during the running of the job. The `restartable` attribute is a Boolean and can be `TRUE` or `FALSE`. The Scheduler will retry a maximum of six runs, and will do so with a growing wait time interval. The first retry is for 1 second after the initial failure. For the second retry, the wait time is multiplied by 10, causing a wait of 10 seconds. For the other retries, the wait time is multiplied by 10 every time until all six retries are passed or failed, or the job finally succeeds. If the job fails for all the retries, the job is marked *broken* and will not be started again until we fix the problems and enable the job again. The Scheduler will stop retrying a job when:
    - The job succeeds
    - All the six retries fail
    - The next retry would make the job retry after the next scheduled normal run

  During the retries, the run count and the failure count are not incremented until the retry stops. At success, the run count is incremented by one and after the final failure, the failure count is incremented by one.

Job execution

It might look like kicking in an open door, but the job owner has automatic execution privileges on his or her own jobs. As `dbms_scheduler` works with `authid current_user`, the executing user also needs the privileges on the objects that are used in the job. Originally, Oracle had stored objects defined with definer's rights. This means that if you have an `execute` permission on a package, with definer's rights (the default), the package can use all the objects that it needs without having to call the user to have privileges on the objects that the package works on. With

`authid current_user`, we run the package with the privileges of the user who calls the package. This means that if (for example) the package wants to insert a row into a table, the calling user needs to not only execute privilege on the package, but also insert privileges on the table. Scheduler objects that can be granted to others are—`job_class` and `Program`. In order to be able to associate a job with a `job_class`, the user must have the `execute` privileges on the job class.

For external jobs, all this is slightly more complicated because the script has its own rules on the machine where it runs. In the database, we can administer everything using Oracle privileges; whereas on the operating system, we have to take into account the way the operating system runs our external job. Normally, the jobs are run by a lower-privileged user on the machine where the database lives. This is administered in `$ORACLE_HOME/rdbms/admin/externaljob.ora`. On the Linux and Unix platforms, this is typically a user, `nobody`, within the `nobody` group. This is not what we want because on many systems, the Oracle software is shared by multiple databases. In that case, running all the jobs on all of the databases using the same operating system account is probably not desirable.

Starting with Oracle version 11g, we can also submit jobs to a remote agent and we can define the credentials to be used to run the job here. We will go into more detail about this in *Chapter 5*, *Getting Out of the Database*. This mechanism with the remote job agent is far smarter than the old external jobs we had to use in 10g. Even when a job runs locally, we should use a remote job agent to run the job because security is now handled in a much smarter way.

When running external jobs, you cannot make any assumption about the environment that the script can use.

Do you remember the results of `TEST03` from Chapter 1? Even the working directory is unusable for us—no `ORACLE_HOME` and no `ORACLE_SID`. When working with Oracle, we need to define both of these to use the software. This is also the reason why many attempts to start Oracle tools directly from the Scheduler fail. Oracle does not support the direct calling of executables and strongly recommends using scripts to initialize the environment at runtime. As no `ORACLE_SID` is specified, I assume that many external job scripts will have `ORACLE_SID` as one of their parameters. One simple example is that of an export. Start with making an operating system script and decide how (and with what parameters) it should be called. Make sure that is also able to run using cron. If the job runs correctly using cron, it is very likely that it also runs correctly using the Scheduler. Cron also has a very limited environment, and so many users have problems getting their scripts running with cron. In the following code, you will see a generic setup that will work for most Unix installations where, by default, the `oraenv`, `coraenv`, and `dbhome` scripts are located

in the `/usr/local/bin` directory. At the start of the script, we make sure that the normal Unix binaries are resolvable using PATH and the Oracle environment script `oraenv` that calls `dbhome`. Most Oracle-related scripts should be able to use this. When site or application specific code is used that is in a different directory, the directory should also be added to the PATH, or those scripts should be called fully qualified.

```
#!/bin/bash
# this script takes ORACLE_SID as argument 1
#              and the full qualified parameterfile as argument 2
PATH=$PATH:/usr/local/bin:/usr/bin
export PATH
ORAENV_ASK=NO
ORACLE_SID=$1
PARFILE=$2
. oraenv
$ORACLE_HOME/bin/exp parfile=$PARFILE
```

Save this script in a known location. There are many possible locations; one choice could be `/usr/local/bin/run_exp.sh`. The location has to be usable and referenced from the job that has to call the script, as shown next.

Now, make an Oracle Scheduler job that calls this script with the two required arguments, `ORACLE_SID` and the full qualified parameter file name. This can be done as follows:

```
BEGIN
  sys.dbms_scheduler.create_job
   (
    job_name           => 'RUN_EXP',
    job_type           => 'EXECUTABLE',
    job_action         => '/usr/local/bin/run_exp.sh',
    start_date         => systimestamp at time zone 'Europe/Amsterdam',
    number_of_arguments => 2,
    enabled            => FALSE
   );
  sys.dbms_scheduler.set_job_argument_value
   (
    job_name             => 'RUN_EXP',
    argument_position    => 1, argument_value => 'ORCL'
   );
  sys.dbms_scheduler.set_job_argument_value
   (
    job_name             => 'RUN_EXP',
    argument_position    => 2,
    argument_value       => '/tmp/exp_parameters.par'
    );
  sys.dbms_scheduler.enable( 'RUN_EXP' );
END;
```

The job is defined in two parts—one part defines the job, and the other part assigns the variables for the job. As this has to be done in two parts, the job has to be defined with `enabled => FALSE` to prevent it from starting before the variables are defined. The location for the parameter file is not where we would normally put these files—for this example, it is a convenient location that exists on most systems, but is a very bad choice for a production environment. Again, the file has to be usable from the Scheduler job and readable by the operating system user who runs the job.

Scheduler management

Managing the Scheduler in the database is a little vague. Most things are defined very clearly, but there is no such thing as the ability to stop or start the Scheduler in a supported way. In the Oracle RDBMS, there is the system privilege `MANAGE SCHEDULER` that enables you to define job classes, windows, and window groups. Setting and reading Scheduler attributes is controlled by this privilege, as is purging the Scheduler logs. The Scheduler attributes are listed in the `ALL_SCHEDULER_GLOBAL_ATTRIBUTE` view. Not all attributes listed here can be modified, and not all Scheduler attributes are listed. The `current_open_window`, for example, is a read-only and changes when the next window opens or the current window closes.

`max_job_slave_processes` can be used to limit the number of processes the Scheduler is allowed to use. The `max_job_slave_processes` parameter cannot be set to `0`. In the earlier versions of Oracle, we could prevent the `dbms_jobs` jobs from running by setting `JOB_QUEUE_PROCESSES` to `0`. In 11g, this parameter (`JOB_QUEUE_PROCESSES`) is deprecated and maintained only for backward compatibility. In fact, there is no parameter that controls the status of the Scheduler at instance startup.

So how can we prevent jobs from running during unplanned maintenance? The simple answer at this moment is: we cannot, not in a supported way. However, it is possible to make a database event trigger that disables the Scheduler during a normal database shutdown. Just set the `SCHEDULER_DISABLED` attribute to `true` as follows:

```
begin
DBMS_SCHEDULER.SET_SCHEDULER_ATTRIBUTE ('SCHEDULER_DISABLED','true');
end;
/
```

It is nice to have this setting. It only fails to help when "unplanned" events occur, which means that it is caused by a database crash. In such a scenario, it is sensible to have the database quiet until the DBA decides that it is safe to run jobs again. Imagine a schedule in which jobs are spawned every second. By the time we disable the Scheduler, several jobs will have started running. For this reason, it is smarter

to define the `on startup` trigger in such a way that it disables the Scheduler on database startup. When everything is working the way it should be, a separate process should enable the Scheduler. The problem is that even in this case jobs can already be started before the disable scheduler could take place.

It makes sense to have a normal `init` parameter that defines the state of the Scheduler after the database startup. In the old-fashioned `dbms_job`, we could do this by setting `JOB_QUEUE_PROCESSES` to `0`, so why not have something similar with the Scheduler? The following code can be used to handle this:

```
CREATE OR REPLACE TRIGGER "SYSMAN"."DISABLE_SCHEDULER" AFTER
STARTUP ON DATABASE begin
dbms_lock.sleep(3); -- The docs say the trigger is executed after
--                  -- the database opens … in 11.1.0.7 it is before.
DBMS_SCHEDULER.SET_SCHEDULER_ATTRIBUTE ('SCHEDULER_DISABLED','true');
end;
```

When the database is open, check the `ALL_SCHEDULER_GLOBAL_ATTRIBUTE` view to see the status after startup:

```
SQL> col value form a10
SQL> select value from ALL_SCHEDULER_GLOBAL_ATTRIBUTE
  2  where attribute_name = 'SCHEDULER_DISABLED';

VALUE
----------
true
```

Don't forget to enable the Scheduler when the system is checked and found to be in a good working state. This can be done in SQL*Plus as follows:

```
SQL> exec DBMS_SCHEDULER.SET_SCHEDULER_ATTRIBUTE ('SCHEDULER_
DISABLED','false');

PL/SQL procedure successfully completed.
```

It is funny to see that this state can be set to `false` multiple times without any error. When the `SCHEDULER_DISABLED` status is `false`, the row no longer exists in the `ALL_SCHEDULER_GLOBAL_ATTRIBUTE` view, so don't be surprised when the query to check the `SCHEDULER_DISABLED` state does not return any row. It just means that the Scheduler is enabled.

As the Scheduler is tightly coupled to the resource manager, it would be fair to have the Scheduler administrator administer the resource manager plans. To arrange all this work neatly, the resource manager plan and the job classes have to be documented and defined very clearly. Developers should try to create jobs and link them to the desired job class.

Logging

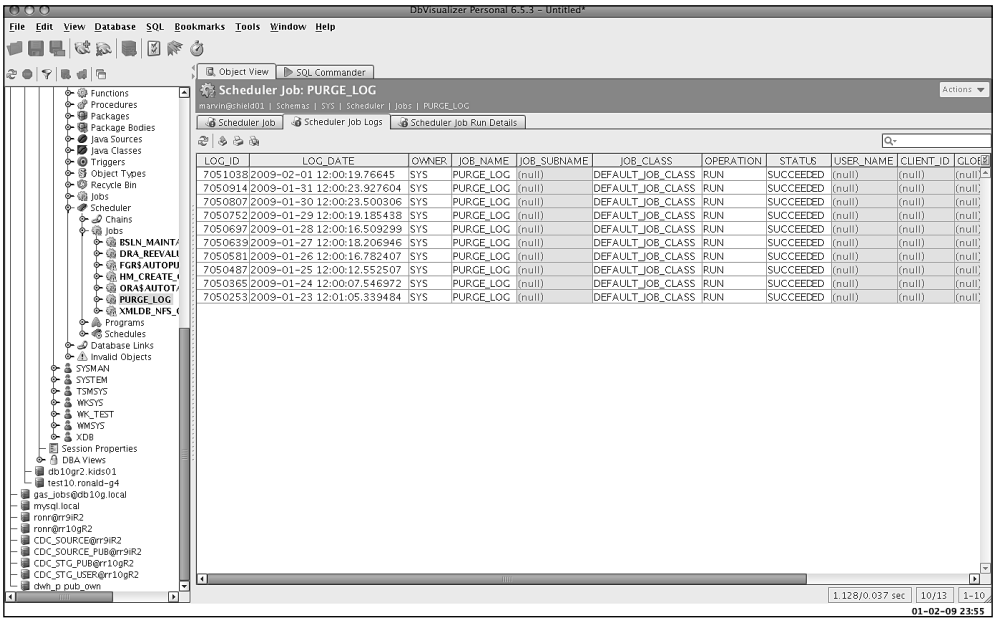
One benefit of `dbm_scheduler` over `dbms_job` is the fact that it records job actions and (if needed) the actions on the jobs. This enables us to find out when jobs ran and compare the runtime behavior based on recorded execution times, instead of vague user estimations and assumptions. Not everything we would wish for is recorded. It is very useful to add some performance metrics to the detailed logging. This can help us spot where the longer job runtime came from. For example, when we see that a job that normally runs for 2 seconds and performs 4,000 buffer gets, now ran for 3 hours and performed 800,000,000 buffer gets, it's clear that some investigation is required.

The disadvantage of this logging is that it has to be configured and maintained. For this, we have the `log_detail_level` and the `log_history` parameters of the job, `job_class`, or the Scheduler.

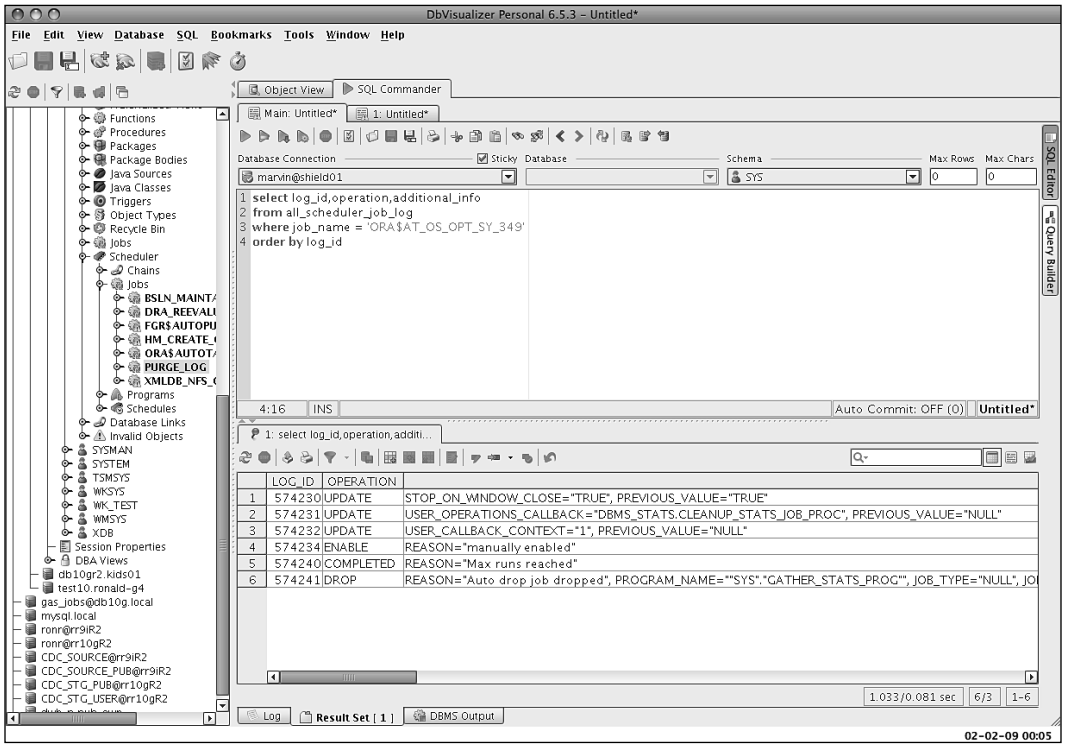
Log detail level

The log detail level can be defined at several locations. It can be defined in the job creation and `job_class`, where the `NULL` value for the job means its `log_detail_level` is inherited from the `job_class` to which the job belongs. Logging can be found in the `ALL_SCHEDULER_JOB_RUN_DETAILS` and `ALL_SCHEDULER_JOB_LOG` views.

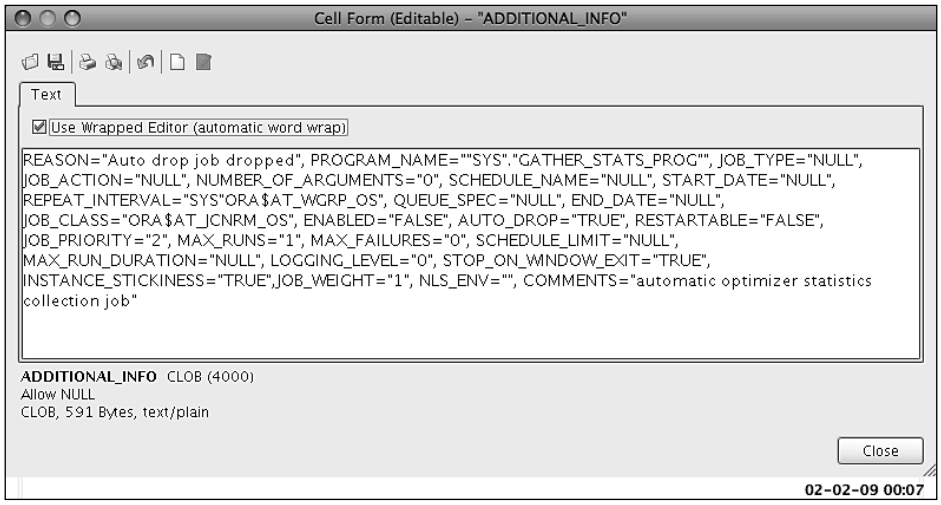
JOB_NAME, **JOB_ID**, and **JOB_CLASS** are recorded (among others) in **Scheduler job Logs**.



In this DbVisualizer screenshot, you can see that this particular job only logs the runs. When job logging is defined with `DBMS_SCHEDULER.LOGGING_FULL`, it logs runs and modifications of the job. The DROP is also in the log. This can be useful when there are jobs that are of the *run once* type and have `auto_drop` set to `true`. These jobs are dropped automatically when they are complete. In that case, we can still find the definition of the dropped job in the `ALL_SCHEDULER_JOB_LOG` view in the **ADDITIONAL_INFO** column. Look at the following screenshot:



We can take a closer look at the **ADDITIONAL_INFO** column where we see `operation = DROP`.



This happens to be one of the jobs that Oracle creates for us to generate optimizer statistics.

In order to see the complete picture, you need to combine both views. Some entries in the log can be a little misleading—especially, the recorded events for chains. These record events look a bit weird as mentioned in *Chapter 2*, *The Simple Chain*. In the log, we will see an operation named `CHAIN_START` with a `RUNNING` status. Normally, we expect to see the `CHAIN_START` operation with a result: `SUCCEEDED` or `FAILED`.

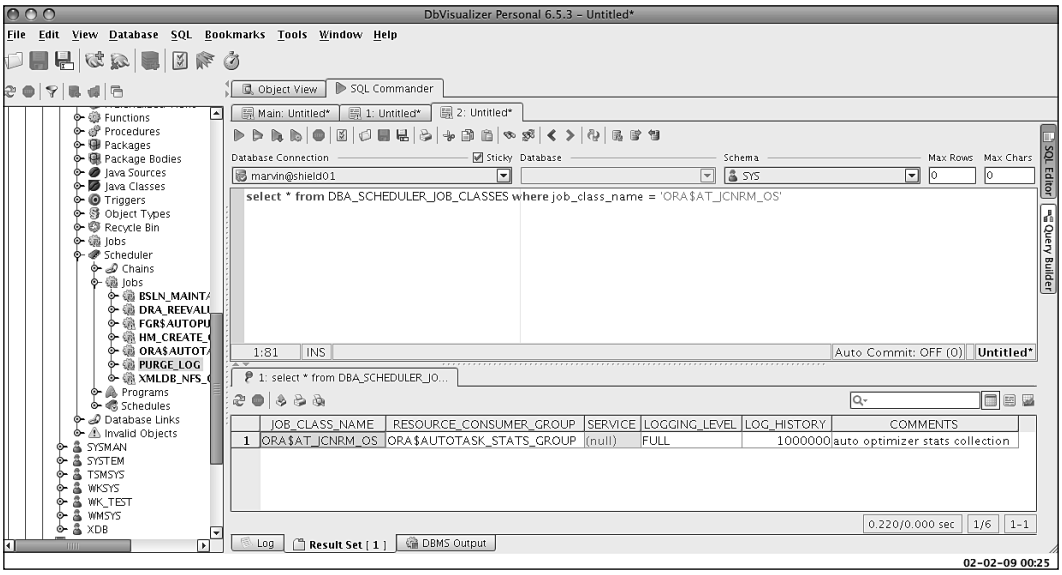
Log purging

As mentioned before, the log purging is controlled by the log history parameters for the job, the job class, or the Scheduler, depending on the level at which `log_history` is first defined. With the default setting of `30 DAYS` and a job running every second, this means quite a lot of rows in the `*_scheduler_job_log` view, possibly 86,400 per job every day. It is possible to run `dbms_scheduler.purge_log` manually, but why bother? Daily at `03:00`, `purge_log job` (from `SYS`) runs `dbms_scheduler.auto_purge`, and it does quite a good job. Sometimes there are complaints about logs that are not purged. A log of a chained job is allowed to be purged only when the chained job is completed. This can pass unnoticed, until one sees logs that are older than expected. In that case, the chain might be stalled, meaning it does not know what to do next based on the defined rules. So, you can force the job to an end. In Chapter 7, *Debugging the Scheduler*, we will see more about making chains run again.

If we want, we can still purge log entries manually using `dbms_scheduler.purge_log`. During development, this can help taking old, failed runs away from the log. Take good care when running it because the default is to clear each and every entry from every log. The following is the definition found in the dbms_scheduler package:

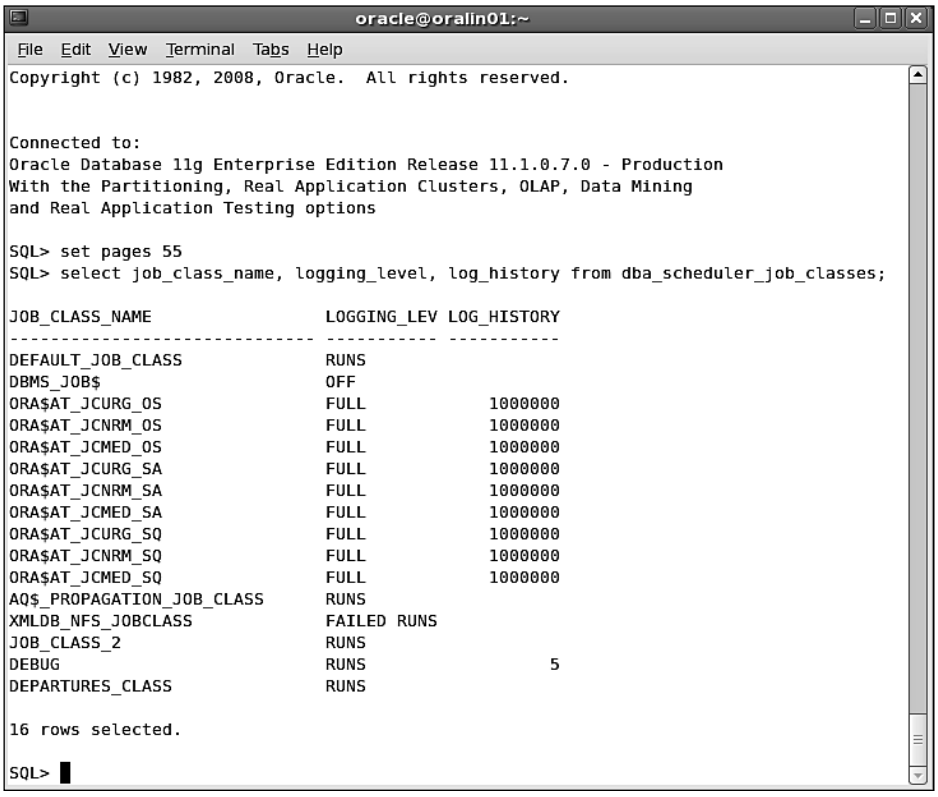
```
-- The following procedure purges from the logs based on the arguments
-- The default is to purge all entries
PROCEDURE purge_log(
  log_history        IN PLS_INTEGER DEFAULT 0,
  which_log          IN VARCHAR2    DEFAULT 'JOB_AND_WINDOW_LOG',
  job_name           IN VARCHAR2    DEFAULT NULL);
```

Many tools use less devastating defaults. The log of the jobs has a value, so why does Oracle have something like "clear all logging" as a default? As always, maintaining logs is something that has to be checked. In this regard, don't forget to check the job class definitions. Did you see the age of the log entries for the `ORA$AT_OS_OPT_SY_349` job? Check out the following screenshot. In the job class for this job (`ORA$AT_JCNRM_OS`), we can see why these types of logs will be kept for a while.



The `log_history` is kept for 1,000,000 days. This must be a reason enough to check all job class definitions as follows:

```
SQL> select job_class_name, logging_level, log_history from
DBA_SCHEDULER_JOB_CLASSES;
```



Just in case you think you can do with less than 1 million days of log history, use the following code that changes the `log_history` to 120 days:

```
begin
  dbms_scheduler.set_attribute('sys.ORA$AT_JCNRM_OS',
                               'log_history',120);
end;
```

Manufacturers of data storage won't like this, because now our databases will not grow as fast as before reducing the `log_history`. When it comes to logging, never take anything for granted. Check the definition of the job and job classes as both can surprise you. Setting a logging level to `full` will cost a little extra disk space, but correcting the `log_history` for the Oracle-created automatic jobs will compensate for that. Having the ability to check back what the job definition was after it was dropped can be very valuable.

Summary

In this chapter, we have seen:

- The privileges needed to create regular jobs
- The privileges needed to create external jobs
- The privileges needed to execute jobs
- The risk of granting the `create any job` privilege
- The privileges needed to do maintenance on the Scheduler system
- Where the logging goes
- How to get rid of the log entries—either manually or automatically
- How to completely disable the Scheduler for maintenance
- How to disable the scheduler on database start
- How the `restartable` job attribute works
- How we can define `job_priority`
- How the `job_priority` is used only with the same `job_class` and `start_date`
- How to use the Schedule limit
- How to specify job over the `max_run_duration` event
- How to use the job log to find the definition of a deleted job
- How to check the log retention
- Which tables to check for growth when using Scheduler logging

In the next chapter, we will be looking at managing resources.

Where to buy this book

You can buy Mastering Oracle Scheduler in Oracle 11g Databases from the Packt Publishing website: `http://www.packtpub.com/mastering-oracle-scheduler-in-oracle-11g-databases/book`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/mastering-oracle-scheduler-in-oracle-11g-databases/book