Sneak Peek Beta 1 Edition

with commentary by
Jonathan Lewis

CHAPTER 1

ORACLE DATABASE 10*g* New Features
Server Manageability

Summary of Contents

Chapter 1 Oracle Database 10*g* Server Manageability
Chapter 2 Performance Tuning New Features
Chapter 3 Security New Features
Chapter 4 Availability New Features
Chapter 5 Business Intelligence Improvements
Chapter 6 Application Development New Features
Chapter 7 Other Database New Features
Chapter 8 Java, JDBC, SQLJ, and Other New Web-Related Features

Welcome to Chapter 1 of *Oracle Database 10*g *New Features*! This chapter first explains how to upgrade to Oracle Database 10*g* and then describes the various Oracle Database 10*g* features that are designed to make managing the server easier. This preview is based on the current beta at the time of writing. The final version of the book will cover many more exciting new manageability features from the final release of the software. The following are the specific topics that are covered:

- Upgrading to Oracle Database 10*g*
- Using new statistics-collection features
- Flushing the database buffer cache
- Using the Database Resource Manager new features
- Firing up the new job scheduler (known as The Scheduler)
- Learning all about user-configurable default tablespaces
- Using tablespace groups and multiple default temporary tablespaces
- Renaming tablespaces
- Dropping databases
- Taking advantage of new LOB storage limitations
- Using the SYSAUX tablespace
- Using Automated Storage Management (ASM)

Upgrading to Oracle Database 10g

Oracle Database 10*g* provides a fairly easy upgrade path for users of older Oracle versions. The following versions can directly be upgraded to Oracle Database 10*g*:

- Oracle Database 8.0.6
- Oracle Database 8.1.7
- Oracle Database 9.0.1
- Oracle Database 9.2

If your database version is not in the preceding list, then you must first upgrade to one of these versions, after which you can upgrade to Oracle Database 10*g*.

Upgrading the Database

After you are at a supported upgrade level, you can upgrade to Oracle Database 10*g* by using any one of the following four upgrade options:

- Use the Oracle Database Upgrade Assistant (DBUA).
- Perform a manual upgrade.

- Use exp/imp to copy the data in your database to a new 10*g* database.
- Use the SQL*Plus **copy** command or the **create table as select** command to copy the data from your current database to your new 10*g* database.

NOTE
Always back up your database before you start your upgrade!

The DBUA

The DBUA is a GUI that is designed for upgrading your Oracle database to Oracle Database 10*g*. You will have the option of starting the DBUA from the Oracle Universal Installer (OUI) when installing Oracle Database 10*g*. DBUA guides you through the upgrade of your Oracle database. You can also start the DBUA at any time in stand-alone fashion (from the command line, just enter **dbua**) to upgrade your database. From Windows, you can also start the DBUA from the Start menu (either from the Oracle folder or use start | run and enter dbua). One nice feature of the DBUA is that it will offer to back up your database for you. This feature does have some limited functionality, because backups to removable media are not supported.

CAUTION
*Oracle Database 10*g *only supports a direct downgrade back to Oracle release 9.2.0.3 or later. You can use imp/exp (Oracle's import/export utilities), however, to move the migrated database data to other versions of Oracle. I strongly advise that you test this method of downgrading on a non-production server first, if you plan to use it.*

Performing Manual Upgrades

Manual upgrades (my personal favorite) allow you to use a series of scripts and utilities to upgrade your database. The summary steps of manual upgrades include:

- Develop a test plan to run after your upgrade.
- Back up your database.
- Run the Upgrade Information Tool (UIT), which is a SQL script, `utlu10li.sql`, located in the directory `$ORACLE_HOME/rdbms/admin`. This script analyzes your database before you upgrade it and alerts you to any problems that might endanger the successful upgrade of your database.
- Upgrade the database. Follow the Oracle upgrade instructions for your specific version and operating system. This step includes the creation of the new SYSAUX tablespace, which is new in Oracle Database 10*g*. (This tablespace is described in detail later, in the section "The SYSAUX Tablespace.")
- Check the component registry (`DBA_REGISTRY`) to make sure your upgrade was successful.
- Back up your new Oracle Database 10*g* database.
- Run your test plan and validate your upgrade.

Jonathan Says…

Remember that **sql_trace** (possibly started by a logon trigger that calls the packaged procedure **dbms_support.start_trace**) is your best friend when you are testing. Many of the enhancements and features that appeared in Oracle9*i* were supported by PL/SQL packages and recursive SQL. Expect more of the same approach to appear in version 10*g*.

If you switch on **sql_trace** when testing a feature, you may find out what Oracle Database 10*g* is doing under the covers to support that feature and whether or not it is suitable for your production system.

Another little trick for discovering hidden costs when you start to test new data structures is to start with a clean schema, create an example of the new data structure, and then query the USER_OBJECTS view to discover what hidden objects Oracle has created to support that structure.

The Compatible Parameter Once you have upgraded to Oracle Database 10*g*, the **compatible** parameter can be set no lower than 9.2.0. Thus, if you are upgrading from 8.0.6, you need to set **compatible** to 9.2.0 before you can open your database under Oracle Database 10*g*. The Oracle Upgrade manual (Oracle10*g* Upgrade Guide) provides detailed instructions on setting the **compatible** parameter. Once you are satisfied that the database can operate under Oracle Database 10*g*, you can set the **compatible** parameter to 10.0. Note that, once you set the **compatible** parameter to 10.0, you cannot set it back. This is different than in previous versions of Oracle. Also note that the command **alter database reset compatibility** is now obsolete.

There are a number of other possible upgrade issues that you will need to deal with depending on the database features that you are using. I strongly suggest that you carefully review the Oracle Upgrade documentation, and that you test your Oracle Database 10*g* upgrades several times before doing one for real in production.

One final upgrade thought. I suggest that you do not use any of the new Oracle Database 10*g* features in a production environment until you have tested the feature thoroughly. While Oracle does its best to regression test new features, there are always a few kinks to be worked out in the beginning. If you find a new feature irresistible (and after you read this book, I hope you do!), then by all means try it out. Test it over and over to make sure it works the way it's intended, and that it doesn't have some nasty impacts, like causing performance problems or causing your database to crash. Also, check Oracle MetaLink, and even open an Oracle iTAR, before you use a new feature that will be a prominent part of your design.

Other Upgrade Methods

The use of the Oracle exp/imp utilities is supported for migrating your Oracle database data to Oracle Database 10*g*. You will use the export utility associated with the version of the database you are currently on (e.g., 8.0.6) to create the dump file. Use the imp utility from the 10*g* Oracle database to import the dump file created for the upgrade. The

Oracle upgrade manual provides a complete set of instructions on how to perform this type of upgrade.

Finally, you can use the SQL*Plus **copy** command or the SQL **create table as select** command to move your database data to a new Oracle Database 10*g* instance. Again, the Oracle upgrade manual provides a complete set of instructions on how to perform this type of upgrade.

Statistics Collection

Oracle Database 10*g* offers some new features to help you collect database statistics. These new features include collection of data dictionary statistics, new behaviors associated with the **dbms_stats** package, and new features related to monitoring tables in the database.

Collecting Data Dictionary Statistics

The Rule Based Optimizer (RBO) is desupported with Oracle Database 10*g*. It's still there in Oracle Database 10*g*, but Oracle is moving away from it quickly and you will find no bug fixes associated with it in future versions of the database. With desupport of the RBO, it becomes even more important to address the question of collection of database statistics.

Oracle Database 10*g* includes new statistics-gathering features. This includes the ability to collect data dictionary statistics, which is now recommended as a best practice by Oracle. Also, Oracle Database 10*g* includes new features that enhance the generation of object level statistics within the database. Let's look at these next.

Data Dictionary Statistics Collection

Oracle Database 10*g* is a big departure from previous releases of Oracle insofar as Oracle recommends that you analyze the data dictionary. You can collect these statistics by using either the **dbms_stats.gather_schema_stats** or **dbms_stats.gather_database_stats** Oracle-supplied procedures, as shown here:

```
Exec dbms_stats.gather_schema_stats('SYS')
```

The **gather_schema_stats** and **gather_database_stats** procedures are not new in Oracle Database 10*g*, but using them to collect data dictionary statistics is new, as are some new parameters that are available with these procedures.

Oracle Database 10*g* also offers two new procedure in the **dbms_stats** Oracle-supplied package. First, the **dbms_stats.gather_dictionary_stats** procedure facilitates analysis of the data dictionary. Second the **dbms_stats.delete_dictionary_ stats** procedure allows you to remove data dictionary stats. Here is an example of the use of the **dbms_stats.gather_dictionary_stats** procedure:

```
exec dbms_stats.gather_dictionary_stats;
```

This example gathers statistics from the SYS and SYSTEM schemas as well as any other schemas that are related to RDBMS components (e.g., OUTLN or DBSNMP).

From a security perspective, any user with SYSDBA privileges can analyze the data dictionary. However, non-SYSDBA user accounts must be granted the **analyze any dictionary** system privilege to be able to analyze the data dictionary.

Gathering Fixed Table Statistics

A new parameter to the **dbms_stats.gather_database_stats** and **dbms_stats.gather_database_stats** supplied procedures is **gather_fixed**. This parameter is set to **false** by default, which disallows statistics collection for fixed data dictionary tables (e.g., x$ tables). Oracle suggests that you analyze fixed tables only once during a typical system workload. You should do this as soon as possible after your upgrade to Oracle Database 10*g*, but again it should be under a normal workload. Here is an example of the use of the **gather_fixed** argument within the **dbms_stats.gather_schema_stats** procedure:

```
Exec dbms_stats.gather_schema_stats('SYS',gather_fixed=>TRUE)
```

Yet another new procedure, **dbms_stats.gather_fixed_objects_stats**, has been provided in Oracle Database 10*g* to collect object statistics on fixed objects. It also has a brother, **delete_fixed_objects_stats**, which will remove the object statistics. Second cousins and new Oracle Database 10*g* provided procedures include **dbms_stats.export_fixed_objects_stats** and **dbms_stats.import_fixed_ objects_stats**. These allow you to export and import statistics to user-defined statistics tables, just as you could with normal table statistics previously. This allows your data dictionary fixed statistics to be exported out of and imported into other databases as required. One other note: the **dbms_stats** Oracle-supplied package also supports analyzing specific data dictionary tables.

When to Collect Dictionary Statistics

Oracle recommends the following strategy with regard to analyzing the data dictionary in Oracle Database 10*g*:

1. Analyze normal data dictionary objects (not fixed dictionary objects) using the same interval that you currently use when analyzing other objects. Use **gather_database_stats**, **gather_schema_stats**, or **gather_dictionary_stats** to perform this action. Here is an example:

   ```
   Exec dbms_stats.gather_schema_stats('SYS',gather_fixed=>TRUE)
   ```

2. Analyze fixed objects only once, unless the workload footprint changes. Generally, use the **dbms_stats.gather_fixed_object_stats** supplied procedure when connected as SYS or any other SYSDBA privileged user. Here is an example:

   ```
   Exec dbms_stats.gather_fixed_objects_stats('ALL');
   ```

New DBMS_STATS Behaviors

Oracle has introduced some new arguments that you can use with the **dbms_stats** package in Oracle Database 10*g*. The **granularity** parameter is used in several **dbms_stats** subprograms (e.g., **gather_table_stats** and **gather_schema_stats**) to indicate the granularity of the statistics that you want to collect, particularly for partitioned tables. For example, you can opt to only gather global statistics on a partitioned table, or you can opt to gather global and partition-level statistics. The **granularity** parameter comes with an **auto** option. When **auto** is used, Oracle collects global, partition-level, and subpartition-level statistics for a range-list partitioned table. For other partitioned tables, only global and partition-level statistics will be gathered.

A second **granularity** option, **global and partition**, will gather the global and partition-level statistics but no subpartition-level statistics, regardless of the type of partitioning employed on the table. Here are some examples of using these new options:

```
Exec dbms_stats.gather_table_stats('my_user','my_tab',granularity=>'AUTO');
Exec dbms_stats.gather_table_stats('my_user','my_tab', -
granularity=>'GLOBAL AND PARTITION');
```

New options are also available with the **degree** parameter, which allows you to parallelize the statistics-gathering process. Using the new **auto_degree** option, Oracle will determine the degree of parallelism that should be used when analyzing the table. Simply use the predefined value, **dbms_stats.auto_degree**, in the **degree** parameter. Oracle will then decide the degree of parallelism to use. It may choose to use either no parallelism or a default degree of parallelism, which is dependent on the number of CPUs and the value of various database parameter settings. Here is an example of the use of the new **degree** option:

```
Exec dbms_stats.gather_table_stats -
('my_user','my_tab',degree=>dbms_stats.auto_degree);
```

Finally, the **stattype** parameter is a new parameter that allows you the option of gathering both data and caching statistics (which is the default) or only data statistics or only caching statistics. Valid options are **all**, **cache**, or **data**, depending on the type of statistics you wish to gather. Here is an example of the use of the **stattype** parameter:

```
Exec dbms_stats.gather_table_stats -
     ('my_user','my_tab',stattype=>'ALL');
```

New Table-Monitoring Behaviors

Oracle9*i* allows you to monitor a table's usage. The monitoring process keeps track of differential changes to the table. The **dbms_stats** package can be used to apply those differential statistics to the table's dictionary statistics, allowing the cost-based optimizer to generate plans based on more current statistics.

Oracle Database 10*g* enables global table monitoring by default. This feature is controlled via the **statistics_level** parameter (**statistics_level** was available in Oracle9*i*). When **statistics_level** is set to TYPICAL (which is the default setting) or ALL, then global monitoring is enabled. When the **statistics_level** parameter is set to BASIC, global monitoring is disabled.

Jonathan Says...

When you start to plan your strategy for gathering statistics, remember this—you probably don't need the same level of statistics collection on every object in the schema. Most (large) objects will give perfectly adequate results from a very small sample size, a few objects will need larger samples, and a handful of columns may need carefully considered histograms.

Although Oracle supplies a **gather_schema_stats** procedure, don't worry if you don't have the time window to use it—you probably don't need to. And remember, if you create loads of unnecessary histograms, you could be creating a performance problem—you pay for histograms through extra memory, CPU, and latch costs during optimization.

Note that if global monitoring is enabled, you cannot disable it for specific tables. The **nomonitoring** clause of the **alter table** command will appear to complete successfully, but will have no effect in reality. Also, the **monitoring** clause of the **alter table** command no longer has any impact on monitoring of tables. It's kind of an all or nothing deal these days!

Flushing the Buffer Cache

Prior to Oracle Database 10*g*, the only way to flush the database buffer cache was to shut down the database and restart it. This was perhaps not the most graceful way of performing this activity because it required shutting down applications and disconnecting users, creating all sorts of mayhem (not that flushing the buffer cache in and of itself can't cause some short-term mayhem of its own!).

Oracle Database 10*g* now allows you to flush the database buffer cache with the **alter system** command using the **flush buffer_cache** parameter, as shown in this example:

```
Alter system flush buffer_cache;
```

NOTE
I am not sure why you would want to flush the buffer cache. If you can think of a good reason why, let me know!

Database Resource Manager New Features

The Database Resource Manager in Oracle Database 10*g* offers a few new features that you need to be aware of:

- The ability to revert to the original consumer group at the end of an operation that caused a change of consumer groups
- The ability to set idle timeout values for consumer groups

- The ability to create mappings for the automatic assignment of sessions to specific consumer groups

Each of these topics is discussed, in turn, in more detail in the following sections.

Reverting Back to the Original Consumer Group

Prior to Oracle Database 10*g*, if a SQL call caused a session to be put into a different consumer group (for example, because a long-running query exceeded a SWITCH_TIME directive value in the consumer group), then that session would remain assigned to the new resource group until it was ended. Oracle Database 10*g* allows you to use the new SWITCH_BACK_AT_CALL_END directive to indicate that the session should be reverted back to the original consumer group once the call that caused it to switch consumer groups (or the top call) is complete.

This is very useful for n-tier applications that create a pool of sessions in the database for clients to share. Previously, after the consumer group had been changed, all subsequent connections would be penalized based on the settings of the consumer group resource plan. The new SWITCH_BACK_AT_CALL_END directive allows the session to be reset, thus eliminating the impact to future sessions. Here is an example of the use of this new feature:

```
EXEC DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(PLAN => 'main_plan',
GROUP_OR_SUBPLAN => 'goonline', COMMENT => 'Online sessions', CPU_P1 => 80,
SWITCH_GROUP => 'ad-hoc', SWITCH_TIME => 3,SWITCH_ESTIMATE => TRUE,
SWITCH_BACK_AT_CALL_END=>TRUE);
```

In this case, I have created a plan directive that is a part of an overall plan called MAIN_PLAN. This particular plan directive is designed to limit the impact of online ad-hoc users (or perhaps applications that are throwing out a great deal of dynamic SQL that's hard to tune) if they issue queries that take a long time (in this example, 3 seconds). This directive causes a switch to a consumer group called ad-hoc, which would likely further limit CPU and might also provide for an overall run-time limit on executions in this particular plan/resource group. Since I have included the SWITCH_BACK_AT_CALL_END directive in this plan directive, the consumer group will revert back to the original plan after the completion of the long-running operation.

Setting the Idle Timeout

Oracle Database 10*g* allows you to limit the maximum time that a session is allowed to remain idle. The **max_idle_time** parameter allows you to define a maximum amount of time that a given session can sit idle, as is shown in the upcoming example. PMON will check the session once a minute and kill any session that has been idle for the amount of time defined in the plan.

```
EXEC DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(PLAN => 'main_plan',
GROUP_OR_SUBPLAN => 'online', max_idle_time=>300,
comment=> ’Set max_idle_time’);
```

Creating Mappings for Automatic Assignment of Sessions to Consumer Groups

The **dbms_resource_manager.set_group_mapping** procedure allows you to map a specific consumer group to a given session based on either login or run-time attributes. These attributes include:

- The username
- The service name
- The client OS username
- The client program name
- The client machine
- The module name
- The module name action

You then have to determine what session attributes you want to map to a given consumer group. In this example, I have mapped the client machine called *tiger* to the resource consumer group LOW_PRIORITY:

```
Exec dbms_resource_manager.set_group_mapping
(DBMS_RESOURCE_MANAGER.CLIENT_MACHINE,‘tiger’,’low_priority’);
```

Thus, if anyone logs in to the database from the machine named tiger, they will be assigned to the consumer group LOW_PRIORITY, which will have already been created.

Often times, there can be a number of mappings that apply to a given session, and a priority has to be defined. This is done by using the procedure **dbms_resource_manager.set_mapping_priority**. This example creates two mappings:

```
Dbms_resource_manager.set_group_mapping
(DBMS_RESOURCE_MANAGER.CLIENT_MACHINE, ‘tiger’,’low_priority’);
Dbms_resource_manager.set_group_mapping
(DBMS_RESOURCE_MANAGER.ORACLE_USER, ‘NUMBER_ONE’,’high_priority’);
```

In this case, anyone signing in from tiger is assigned to the LOW_PRIORITY consumer group, but where will the user NUMBER_ONE be assigned? Well, right now it’s hard to tell. So, to make sure that NUMBER_ONE is always set to be assigned to the high-priority resource consumer group, I can use the provided procedure called **dbms_resource_manager.set_mapping_priority**:

```
Dbms_resource_manager.set_mapping_priority(ORACLE_USER=>1,
CLIENT_MACHINE=>2, EXPLICIT=>3, MODULE_NAME=>4, SERVICE_NAME=>5,
CLIENT_OS_USER=>6, CLIENT_PROGRAM=>7, MODULE_NAME_ACTION=>8);
```

This code will cause Oracle to prioritize consumer group selection based first on username and then on the client machine name. So, now the user NUMBER_ONE will always get the higher-priority consumer group assignment.

Be aware that regardless of consumer group assignments, a user must still be given switching privileges into a given consumer group. If the user has not been granted such privileges, then sessions will not be switched.

Scheduler Changes

Oracle Database 10*g* offers a brand new job-scheduling facility, known as The Scheduler. The Scheduler is controlled via the new Oracle Database 10*g* supplied package **dbms_scheduler**. This package replaces the **dbms_job** package that has been around for some time.

Overview of The Scheduler

The new scheduler offers much added functionality over the **dbms_job** package. The Scheduler enables you to execute a variety of stored code (such as PL/SQL), a native binary executable, and shell scripts. The object that is being run by The Scheduler is known as the *program.* The program is more than just the name; it includes related metadata about the program, such as the arguments to be passed to it and the type of program that is being run.

Different users can use a program at different times, eliminating the need to have to redefine the program every time you wish to schedule a job. Programs can be stored in *program libraries,* which allows for easy reuse of program code by other users.

Each program, when scheduled, is assigned to a *job.* A job can also just contain an anonymous PL/SQL block instead of a program. The job is a combination of the program (or anonymous PL/SQL block) and the schedule associated with the program, which defines when the job is to run. Also associated with the job is other metadata related to the job, such as the job class and the window or window group.

The *job class* is a category of jobs that share various characteristics, such as resource consumer group assignments and assignments to a common, specific, service name. The job class is related to the job window.

The job *window*, or *window group,* essentially allows the job to take advantage of specific resource plans. For example, if the schedule for a job is for it to run every hour, the job window will allow it to run under one resource group in the morning and a different resource group in the evening. That way, you can control the resources the job can consume at different times throughout the day.

Oracle provides two different interfaces into The Scheduler. The first is the **dbms_scheduler** package and the second is through the Oracle Enterprise Manager (OEM).

Practical Use of The Scheduler

There are a few steps to follow when you want to assign a job to The Scheduler:

- Create the program (optional).
- Create the job.

Creating a Program in The Scheduler

Creating a program is the optional first step when creating a scheduled operation. This operation may actually take four steps:

1. Create the program itself.
2. Define the program arguments.
3. Create the job.
4. Define job arguments.

The following sections explain each of these steps in turn.

Creating the Program To create a program, so that you can schedule it, you use the PL/SQL-supplied procedure **dbms_scheduler.create_program**. To use this package in your own schema, you must have the **create job** privilege. To use it to create jobs in other schemas, you need the **create any job** privilege. By default, a program is created in a disabled state (which can be overridden by setting the **enabled** parameter of the **create_program** procedure to TRUE). First, let's look at the definition of the **dbms_scheduler.create_program** procedure:

```
DBMS_SCHEDULER.CREATE_PROGRAM (
program_name IN VARCHAR2,
program_type IN VARCHAR2,
program_action IN VARCHAR2,
number_of_arguments IN PLS_INTEGER DEFAULT 0,
enabled IN BOOLEAN DEFAULT FALSE,
comments IN VARCHAR2 DEFAULT NULL);
```

It always helps to know what the various parameters are for, of course. So let's look at a description of the parameters for the **create_program** procedure:

Parameter Name	Description
program_name	Identifies the name of the program. This is an internally assigned name, which represents the **program_action** that will be executed.
program_type	Identifies the type of executable being scheduled. Currently, the following are valid values: PLSQL_BLOCK, STORED_PROCEDURE, and EXECUTABLE.

Parameter Name	Description
program_action	Indicates the procedure, executable name, or PL/SQL anonymous block associated with the program.
number_of_arguments	Identifies the number of arguments required for the program (ignored if **program_type** is PLSQL_BLOCK).
Enabled	Indicates whether the program should be enabled when created.
Comments	Allows freeform comments describing the program or what it does.

Here are some examples of the creation of programs:

```
BEGIN
 dbms_scheduler.create_program(
  program_name => 'delete_records',
  program_action => '/opt/oracle/maint/bin/nightly_delete_records.sh',
  program_type => 'EXECUTABLE', number_of_arguments=>2);
END;
```

In this example, I am creating a program called `delete_records`. It is an external executable, a shell script in this case. The program is located in `/opt/oracle/maint/bin` and called `nightly_delete_records.sh`. Note that Oracle does not check for the existence of the program when the **create_program** procedure is executed. Thus, you can create your program even if the underlying executable doesn't exist.

You can create a program for an anonymous PL/SQL block as well, as demonstrated in this example:

```
BEGIN
     dbms_scheduler.create_program(
         program_name => 'sp_delete_records',
         program_action => 'DECLARE
                                 rec_count   number;
                            BEGIN
                            DELETE FROM old_records
                            WHERE record_date < sysdate - 5;
                            rec_count:=sqlcommand%ROWCOUNT;
                            insert into records_removed
                                 (date, table, how_many, job_ran) VALUES
                                 (sysdate, 'OLD_RECORDS', rec_count,
                                  scheduler$_job_start);
                            COMMIT;
                            END;',
         program_type => 'EXECUTABLE');
END;
```

In the case of this anonymous block, I used one of several supplied special variable names in my code (in this case, scheduler$_job_start). These variables are described briefly in the following table:

Variable Name	**Description**
scheduler$_job_name	Provides the name of the job being executed
scheduler$_job_owner	Provides the name of the owner of the job
scheduler$_job_start	Provides the start time of the job
scheduler$_window_start	Indicates the start time of the window associated with the job
scheduler$_window_end	Indicates the end time of the window associated with the job

OEM also provides an interface to create programs that you can use if you prefer that method.

You can drop a program with the **dbms_scheduler.drop_program** procedure, as shown in this example:

```
Exec dbms_scheduler.drop_program('delete_records');
```

Defining the Program Arguments Many programs have arguments (aka parameters) that need to be included when that program is called. You can associate arguments with a program by using the **dbms_scheduler.define_program_ argument** procedure. Using the previous program example, `delete_records`, I can add some arguments to the program as follows:

```
BEGIN
dbms_scheduler.define_program_argument(
program_name => 'delete_records',
argument_name => 'delete_date',
argument_position=>1, argument_type=>'date',
default_value=> 'to_char(sysdate - 5, ''mm/dd/yyyy'')'  );
end;
/
```

To be able to call this program, you need the **alter any job** or **create any job** privilege. Additionally, calling this problem does not change the state of the associated job (enabled or disabled). You can replace an argument by simply calling the **define_ program_argument** procedure and replacing an existing argument position.

Creating the Job

To actually get The Scheduler to do something, which is kind of the idea, you need to create a job. The job can either run a program that you have created (refer to the previous section) or run its own job, which is defined when the job is defined. The job consists of these principle definitions:

- **The schedule** This is when the job is supposed to do whatever it's supposed to do. The schedule consists of a start time, an end time, and an expression that indicates the frequency of job repetition.
- **The associated job argument (or the what)** This is what the job is supposed to do. This can be a pre-created PL/SQL or Java program, anonymous PL/SQL, or even an external executable (for example, a shell script or C program call).
- **Other metadata associated with the job** This includes such things as the job's class and priority, job-related comments, and the job's restartability.

Jobs are created with the **dbms_scheduler.create_job** package, as shown in this example:

```
Exec dbms_scheduler.create_job(
     job_name=>'CLEAR_DAILY',
     job_type=>'STORED_PROCEDURE',
     job_action=>'JOBS.SP_CLEAR_DAILY',
     start_date=>NULL,
     repeat_interval=>'TRUNC(SYSDATE) + 1/24',
     comments=>'Hourly Clearout Job');
```

This example creates a scheduled job that executes immediately and then will run every hour thereafter. This job is assigned a name called CLEAR_DAILY. When The Scheduler runs the job, a PL/SQL stored procedure called **sp_clear_daily** is executed.

Perhaps another example is in order. In this case, I will create a scheduled job that fires off an external shell script:

```
Exec dbms_scheduler.create_job(
     job_name=>'RUN_BACKUP',
     job_type=>'EXECUTABLE',
     job_action=>'/opt/oracle/admin/jobs/run_job.sh',
     start_date=>'to_date('04-30-2003 20:00:00','mm-dd-yyyy hh24:mi_ss'),
     repeat_interval=>'TRUNC(SYSDATE) + 23/24',
     comments=>'Daily Backup');
```

The **repeat_interval** attribute defines how often and when the job will repeat. If the **repeat_interval** is NULL (the default), the job executes only one time and then is removed. When determining the interval, you have two options. First, you can use the older PL/SQL time expressions for defining the program execution intervals.

Oracle Database 10*g* now offers a new feature, Calendar Expressions, which you can use in lieu of the old PL/SQL time expressions. There are three different types of components: the frequency (which is mandatory), the specifier, and the interval. Frequencies indicate how often the job should run. The following frequencies are available:

Yearly	Monthly	Weekly	Daily
Hourly	Minutely	Secondly	

Additional parameters, the specifier and interval, define in more detail how frequently the job should run.

Defining the Job Arguments If you are scheduling a job that is not associated with a program, then that job may be a program that accepts arguments. If this is the case, you need to use the **dbms_scheduler.set_job_argument_value** procedure. Executing this procedure will not enable or disable any given job. Here is an example of setting some parameters for a job. In this case, I am indicating to the RUN_BACKUP job that it should include an argument of 'TABLESPACE USERS', which might indicate that the backup job should back up the users tablespace.

```
exec dbms_scheduler.set_job_argument_value
( job_name =>'RUN_BACKUP',
  argument_name=>'BACKUP_JOB_ARG1',
  argument_value=>'TABLESPACE USERS');
```

Other Job Scheduler Functionality

The new job scheduler also allows you to define *job classes*, which allow you to define a category of jobs that share common resource usage requirements and other characteristics. One job can belong to only one job class, though you can change the job class that a given job is assigned to. Any defined job class can belong to a single resource consumer group, and to a single service at any given time.

Job classes, then, allow you to assign jobs of different priorities. For example, administrative jobs (such as backups) might be assigned to an administrative class that is assigned to a resource group that allows for unconstrained activity. Other jobs, with a lesser priority, may be assigned to job classes that are assigned to resource groups that constrain the overall operational overhead of the job, so that those jobs do not inordinately interfere with other, higher-priority jobs. Thus, job classes help you to manage the amount of resources that a given job can consume.

To create a job class, you use the **dbms_scheduler.create_job_class** procedure. All classes belong to the SYS schema, and to create one requires the **manage scheduler** privilege. Here is an example of defining a job class:

```
exec dbms_scheduler.create_job_class(
job_class_name=>'CLASS_ADMIN',
resource_consumer_group=>'ADMIN_JOBS',
service=>'SERVCE_B');
```

This job class will be called CLASS_ADMIN. It is assigned to a resource consumer group (that will have already been created) called ADMIN_JOBS, which will no doubt give administrative jobs pretty unfettered access to resources. This job class is also assigned to a specific service, SERVICE_B, so the administrator can define which service the job class is associated with.

Once the job class is defined, you can define which jobs are members of that class when you create the jobs. Alternatively, you can use the **dbms_scheduler.set_ attribute** procedure to assign an existing job to that class.

User-Configurable Default Tablespaces

Oracle offers user-configurable default tablespaces in Oracle Database 10*g*. Once you configure a default user tablespace, all new users will be assigned to that tablespace rather than the SYSTEM tablespace. At the time this was written, this feature was not available to test, but I thought you would like to know it's coming.

Tablespace Groups and Multiple Default Temporary Tablespaces

Oracle Database 10*g* now allows you to define tablespace groups, which are logical groupings of tablespaces. This further allows you to assign temporary tablespaces to those groups, and then assign this tablespace group as the default temporary tablespace for the database. In essence, tablespace groups allow you to combine temporary tablespaces into one tablespace pool that is available for use to the database.

Assigning Temporary Tablespaces to Tablespace Groups

You can assign a temporary tablespace to a tablespace group in one of two ways. First, you can assign it to a tablespace group when you create the tablespace via the **create tablespace** command. Second, you can add a tablespace to a tablespace group via the **alter tablespace** command. An example of each of these operations is listed next (note that OMF is configured in this example):

```
Create temporary tablespace temp_tbs_01 tablespace group tbs_group_01;
alter tablespace temp_tbs_01 tablespace group tbs_group_02;
```

There is no limit to the number of tablespaces that can be assigned to a tablespace group. The tablespace group shares the same namespace as normal tablespaces, so tablespace names and tablespace group names are mutually exclusive. You can also remove a tablespace from a group by using the **alter tablespace** command and using empty quotes as an argument to the **tablespace group** parameter, as shown in this example:

```
Alter tablespace temp3 tablespace group '';
```

Jonathan Says…

Given the introduction of "bigfile" tablespaces (with a maximum size of 8 exabytes, or roughly 8 million terabytes), you have to wonder if there is something more subtle going on here than the declared intention of making more space available for sorting, etc. So, if having multiple tablespaces is good for temporary space, are there some types of systems whose characteristic activity means they should not use "bigfile" tablespaces?

Defining a Tablespace Group as the Default Temporary Tablespace

After you have created the tablespace group and assigned a set of tablespaces to that group, you can assign that group of temporary tablespaces (or that tablespace group) as the default temporary tablespace for the system, or as a temporary tablespace group for specific users.

You can do this in the **create database** statement when you create the database, or you can use the **alter database** statement to modify the temporary tablespace settings. Using either statement, you simply define the tablespace group as the default tablespace, as shown in this example:

```
Alter database default temporary tablespace tbs_group_01;
```

This has the effect of assigning multiple tablespaces as the default temporary tablespace. Once you have assigned a tablespace group as the default temporary tablespace group, you cannot drop any tablespace in that group.

So, now you can define more than a single tablespace as the database default temporary tablespace; as a result, larger SQL operations can use more than one tablespace for sort operations, thereby reducing the risk of running out of space. This also provides more tablespace space, and potentially better I/O distribution for sort operations and parallel slave operations that use temporary tablespaces. If a tablespace group is defined as the default temporary tablespace, then no tablespaces in that group can be dropped until that assignment has been changed.

You can assign a user to a tablespace group that might not be the default tablespace group either in the **create user** or **alter user** statements, as shown in these examples that assign the TBS_GROUP_01 tablespace to the user NO_PS:

```
Create user no_ps identified by gonesville
default tablespace dflt_ts temporary tablespace tbs_group_01;

alter user no_ps temporary tablespace tbs_group_02;
```

Tablespace Group Data Dictionary View

A new view, DBA_TABLESPACE_GROUPS, is available to associate specific temporary tablespaces with tablespace groups. The TEMPORARY_TABLESPACE column of the *_users views will report either the temporary tablespace name or the temporary tablespace group name that is assigned to the user. Here is an example of a query that joins the DBA_USERS and DBA_TABLESPACE_GROUPS views together and gives you a list of users who are assigned a tablespace group as their temporary tablespace name, and all of the tablespaces that are associated with that group:

```
Select a.username, a.temporary_tablespace, b.tablespace_name
from dba_users a, dba_tablespace_groups b
Where a.temporary_tablespace in (select distinct group_name from
dba_tablespace_groups);
```

Renaming Tablespaces

You have been asking for it, I have been asking for it, and now it's here! Oracle Database 10*g* includes the ability to rename tablespaces. You use the **alter tablespace** command with the **rename to** parameter, as shown in this example:

```
Alter tablespace production_tbs rename to prod_tbs;
```

Note that you cannot rename the system tablespace or the SYSAUX tablespace (which is described later in this chapter). Another nice feature is that if the tablespace is an UNDO tablespace, and you are using a server parameter file (SPFILE), Oracle will change the UNDO_TABLESPACE parameter in the SPFILE to reflect the new UNDO tablespace name.

The ability to rename tablespaces has some great practical applications with operations such as transportable tablespaces. Now, rather than having to drop the existing tablespace before you can transport it in, you only need rename that tablespace. Way to go Oracle!

Something to be aware of is that renaming a tablespace does not change the name of the datafile in any way. For example, OMF uses the name of the tablespace (or part of it) in the OMF datafile naming scheme, and frequently DBAs do the same when they manually create a tablespace datafile. Renaming the tablespace will result in the datafiles no longer reflecting the true name of the tablespace.

CAUTION
*You should back up the control file as soon as possible after renaming tablespaces within the database. If you do not, depending on when the backup of the control file took place, a divergence may exist between the tablespace names in the control file and the actual tablespace names in the database. Refer to the Oracle Database 10*g *documentation for more details on specific recovery scenario responses.*

Dropping Databases

The **drop database** command can be used to drop your database. Oracle will drop the database, deleting all control files and all datafiles listed in the control file. If you are using a SPFILE, then Oracle will remove it as well. Only a user with SYSDBA privileges can issue the statement and the database must be mounted (not open) in exclusive and restricted mode. Here is an example of the use of the **drop database** command:

```
Drop database;
```

Larger LOBs

If you use LOBs in your database (NCLOB, BLOB, or CLOB), then you will be happy to know that the limits on LOBs have been increased in Oracle Database 10*g*. The new maximum limits are calculated at (4GB – 1 byte) * (the database block size). Thus, if the database block size is 8KB, there is essentially a 32GB limitation on LOBs in that database. Note that Bfiles are limited to 4GB in size. Load 'em up folks, its ready to rumble!

The SYSAUX Tablespace

The SYSAUX tablespace is a new feature and required component in Oracle Database 10*g*. This section first discusses the SYSAUX tablespace and then reviews some Oracle-supplied procedures that allow you to perform maintenance tasks on the SYSAUX tablespace.

Introducing the SYSAUX Tablespace

The SYSAUX tablespace is a secondary tablespace for storage of a number of database components that were previously stored in the SYSTEM tablespace. It is created as a locally managed tablespace using automatic segment space management.

Previously, many Oracle features required their own separate tablespaces (such as the RMAN recovery catalog, Ultra Search, Data Mining, XDP, and OLAP). This increases the management responsibility of the DBA. The SYSAUX tablespace consolidates these tablespaces into one location, which becomes the default tablespace for these Oracle features.

When you create an Oracle database, Oracle creates the SYSAUX tablespace for you by default. If you are using OMF, then the tablespace is created in the appropriate OMF directory. If you use the **sysaux datafile** clause in the **create database** statement, then the SYSAUX tablespace datafile(s) will be created in the location you define. Finally, if no **sysaux datafile** clause is included and OMF is not configured, Oracle creates the SYSAUX tablespace in a default location that is OS-specific. Here is an example of a **create database** statement with the **sysaux datafile** clause in it:

```
CREATE DATABASE my_db
DATAFILE 'c:\oracle\oradata\my_db\my_db_system_01.dbf' SIZE 300m
SYSAUX DATAFILE 'c:\oracle\my_db\my_db_sysaux_01.dbf' SIZE 100m
DEFAULT TEMPORARY TABLESPACE dtemp_tbs tempfile
'c:\oracle\my_db\my_db_temp_01.dbf' SIZE 100m
```

```
UNDO TABLESPACE undo_tbs_one DATAFILE
'c:\oracle\my_db\my_db_undo_tbs_one_01.dbf' SIZE 100m;
```

As stated earlier in this chapter, when you migrate to Oracle Database 10*g*, you need to create the SYSAUX tablespace as a part of that migration. You do this after mounting the database under the new Oracle Database 10*g* database software. Once you have mounted it, you should open the database in migrate mode with the **startup migrate** command. Once the database is open, you can create the SYSAUX tablespace. Here is the **create tablespace** statement that you would use to perform this operation:

```
CREATE TABLESPACE sysaux

DATAFILE 'c:\oracle\oradata\my_db\my_db_sysaux_01.dbf' SIZE 300m
EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO;
```

The SYSAUX tablespace must be created with the attributes shown in the preceding example. The following restrictions apply to the usage of the SYSAUX tablespace in Oracle Database 10*g:*

- When migrating to Oracle Database 10*g*, you can create the SYSAUX tablespace only when the database is open in migrate mode.
- Also, when migrating to Oracle Database 10*g*, if a tablespace is already named SYSAUX, you will need to remove it or rename it while you are in migrate mode.
- Once you have opened your Oracle Database 10*g* database, you cannot drop the SYSAUX tablespace. If you try, an error will be returned.
- You cannot rename the SYSAUX tablespace during normal database operations.
- The SYSAUX tablespace cannot be transported to other databases via Oracle's transportable tablespace feature.

Once the SYSAUX tablespace is in place and the database has been upgraded, you can add or resize datafiles associated with a SYSAUX tablespace just as you would any other tablespace through the **alter tablespace** command, as shown in this example:

```
ALTER TABLESPACE sysaux ADD DATAFILE
 'd:\oracle\oradata\my_db\my_db_sysaux_01.dbf' SIZE 200M;
```

Managing Occupants of the SYSAUX Tablespace

Each set of application tables within the SYSAUX tablespace is known as an *occupant.* Oracle provides some new views to help you monitor space usage of occupants within the SYSAUX tablespace and some new procedures you can use to move the occupant objects in and out of the SYSAUX tablespace.

First, Oracle provides a new view, V$SYSAUX_OCCUPANTS, to manage occupants in the SYSAUX tablespace. This view allows you to monitor the space usage of occupant application objects in the SYSAUX tablespace, as shown in this example:

```
SELECT occupant_name, space_usage_blocks FROM v$sysaux_occupants;
```

In this case, Oracle will display the space usage for the occupants, such as the RMAN recovery catalog.

If you determine that you need to move the occupants out of the SYSAUX tablespace, then the MOVE_PROCEDURE column of the V$SYSAUX_OCCUPANTS view will indicate the procedure that you should use to move the related occupant from the SYSAUX tablespace to another tablespace. This can also be a method of "reorganizing" your component object tables, should that be required.

Automated Storage Management

Oracle Database 10*g* introduces Automated Storage Management (ASM), a service that provides management of disk drives. ASM can be used on a variety of configurations, including Oracle9*i* RAC installations. ASM is an alternative to the use of raw or cooked file systems. ASM offers a number of features, including:

- Simplified daily administration
- The performance of raw disk I/O for all ASM files
- Compatibility with any type of disk configuration, be it JBOD or complex SAN
- Use of a specific file-naming convention to name files, enforcing an enterprise-wide file-naming convention
- Prevention of the accidental deletion of files, since there is no file system interface and ASM is solely responsible for file management
- Load balancing of data across all ASM managed disk drives, which helps improve performance by removing disk hot spots
- Dynamic load balancing of disks as usage patterns change and when additional disks are added or removed
- Ability to mirror data on different disks to provide fault tolerance
- Support of vendor-supplied storage offerings and features
- Enhanced scalability over other disk-management techniques

ASM can work in concert with existing databases that use raw or cooked file systems. You can choose to leave existing file systems in place or move the database datafiles to ASM disks. Additionally, new database datafiles can be placed in either ASM disks or on the preexisting file systems. Databases can conceivably contain a mixture of file types, including raw, cooked, OMF, and ASM (though the management of such a system would be more complex).

The details of implementing and managing ASM are significant and would consume more than a few chapters. Review the Oracle Database 10*g* documentation for more details on this new Oracle feature.