CHAPTER 13

Performing Flashback Recovery

In Oracle Database 10g Release 1, Oracle introduced a new feature: flashback. Actually, the term is used in three different contexts, making the usage somewhat confusing. There is also a similar-sounding feature—flashback queries—starting with Oracle9i Database, and this doesn't help matters much.

The flashback concepts introduced in Oracle Database 10g are different from the ones introduced in Oracle9i Database. The 10g version of the term—flashback—actually refers to an aid to recoverability. In this chapter, you will learn about the various flavors of flashback and how to use each one. You will also learn how to recover a specific table from the backups without having to recover an entire database.

Introducing Flashback

There are four flavors of flashback:

- Flashing back a database
- Undropping a table
- Flashing back a table
- Recovering a single table from backup

Flashing Back a Database

Many times you might need to roll back the database to a point in time in the past. In earlier releases, this functionality was present but in a very different way. For instance, in Oracle9i Database and prior versions, you would reinstate an older backup and then roll it forward to a point in time in the past. For instance, suppose today is January 21 and the time is 10 a.m. The database backups are taken at 5 a.m. every day. If you wanted to roll the database back to 10 p.m. January 20, you would restore the backup taken at 5 a.m. January 20 and then apply all the archived redo logs to restore it to 10 p.m. But what if you made a mistake in your calculation and the data you wanted to recover was deleted at 9 p.m.? After you recovered the database up to 10 p.m., all your work was in vain. There is no going back to 9 p.m.; you would have to start the process again from the beginning—restore the backup of 5 a.m. and then roll forward all the changes by applying logs up to 9 p.m.

Oracle Database 10g changed all that by introducing a feature called flashback database. You enable flashback on the database. This causes additional logs to be created during the database operation. These logs, called flashback logs, are generated along with the regular archived logs. The flashback logs record changes to the database blocks exclusively for the purpose of rolling back the block changes, so they are different from archived logs. When you flash the database back to a point in the past, these flashback logs are read and applied to the database blocks to undo the changes. The entire database is transported to that point in time.

■ **Note** The entire database is rolled back during flashback database; you can't perform flashback on individual tables or tablespaces.

Undropping a Table

This has happened to the best of us—you dropped a very important table. What can you do? In versions prior to Oracle Database 10g Release 1, there wasn't any choice other than to restore the backup of the corresponding tablespace to another database, recover the tablespace, export the table, and import the table to the production database. These tasks are time-consuming and risky, and the table is unavailable throughout them.

Starting with Oracle Database 10g, the process is less threatening. When the table is dropped, it's not really erased from the database; rather, it is renamed and placed in a logical container called the recycle bin, similar to the Recycle Bin found in Windows. After you realize the mistake, you can reinstate the dropped table using only one simple command. Who says you can't revive the dead?

Flashing Back a Table

The Oracle9i Database introduced a feature called flashback query. When the data in the database changes, the past images of the changed data are stored in special segments called undo segments. The reason for storing this data is simple—if the changed data is not committed yet, the database must reconstruct the prechange data to present a read-consistent view to the other users selecting the same data item. When the change is committed, the need for the past image is gone, but it's not discarded. The space is reused if necessary. The reason for keeping it around is simple, too.

The read-consistency requirement does not stop after the data changes are committed. For instance, a long-running query needs a read-consistent view when the query started, which could be well in the past. If the query refers to the data that might have been changed and committed in the recent past, after the query has started, the query must get the past image, not the image now, even though it is committed. If the query finds that the past data is no longer available, it throws the dreaded `ORA-1555 Snapshot Too Old` error.

Anyway, what does the `ORA-1555` error have to do with flashback operation? Plenty. Since the past image of the data is available in the undo segment for a while, why should a long-running query be the only one to have fun? Any query can benefit from that past image, too. That thought gave rise to flashback queries in Oracle9i Database where you could query data as of a time in the past. With Oracle Database 10g, that functionality was made richer with flashback version queries, where you can pull the changes made to the row data from the undo segments, as long as they are available in the undo segments, of course. And when you pull the older versions of the table, you can effectively reinstate the entire table to a point in time in the past using these past images. This is known as flashing back the table.

Recovering a Table

Suppose you have dropped a table. Now it is gone from the recycle bin, so you can't get it out from there using the undrop process mentioned earlier in this section. The only option is to get it from the backup. However, database recovery recovers the entire database from backup and will overwrite *all* data as of that point in time in the past. When all you want is to recover a single table from the backup, you can't just surgically extract the table from the backup. Instead, you have to create a temporary instance and recover the tablespace containing the table to the very point just prior to the time the table was dropped. However, you need to restore and recover some other mandatory tablespaces, such as system, sysaux, and one or more undo tablespaces. Once they are restored, you will need to recover that temporary database to the required time, extract the table using Data Pump, and import it into the main database. All this is a lot of work.

Oracle Database 12.1 introduces a new feature to recover a single table, or even a single partition of a table from the backup. A single command accomplishes that objective. The job of recovering a single table is now much easier than before.

In this chapter, you will learn to use all four types of flashback operations.

13-1. Checking the Flashback Status of a Database

Problem

You want to check whether your database is flashback enabled.

Solution

The data dictionary view V$DATABASE contains information about the flashback status of the database. Check the column FLASHBACK_ON on that view to ascertain the flashback status:

```
SQL> select flashback_on from v$database;

FLASHBACK_ON
------------
YES
```

The output shows that the value of the column FLASHBACK_ON is set to YES, which means the database is running in flashback mode. If the database is not in flashback mode, the query would have returned NO.

■ **Note** Starting with Oracle 12.1, you can run SQL statements from the RMAN prompt, as well as from the SQL*Plus prompt. For simplicity and consistency with older releases, we execute SQL commands from SQL*Plus in this book.

How It Works

If the database is running in flashback mode, it generates additional files known as flashback logs, which record changes to the data blocks. These files are recorded in the fast recovery area, which was described in Chapter 3. In addition, the FLASHBACK_ON column will return a YES indicator so that you know for sure that flashback mode is enabled.

13-2. Enabling Flashback on a Database

Problem

You want to enable a database to flash back to a point in time in the past.

Solution

The database must be running in archivelog mode to enable flashback. The flashback-enabled database generates flashback logs, which are stored only in the fast recovery area (FRA), which used to be called Flash Recovery Area prior to Oracle Database 11g Release 2. So the FRA must be configured prior to enabling the flashback.

These flashback logs are generated in addition to the archived logs. Here are the steps to then follow to enable flashback on the database:

1. Make sure the FRA is defined in the database. To set up the FRA, check out Recipe 3-1. To find out whether the FRA is set, execute the following command via SQL*Plus while logged in as `sys` or any other `sysdba` account:

   ```
   SQL> show parameter db_recovery_file_dest
   ```

   If the value of the parameter `db_recovery_file_dest` is set, then the FRA is defined to that location. Here is a sample output:

   ```
   SQL> show parameter db_recovery_file_dest

   NAME                                 TYPE        VALUE
   ------------------------------------ ----------- -----
   db_recovery_file_dest                string      +FRA
   db_recovery_file_dest_size           big integer 12G
   ```

2. From the output, you'll notice that the parameter `db_recovery_file_dest` is set to `+FRA`, which is the location of the fast recovery area. The second parameter, `db_recovery_file_dest_size`, shows the size of the fast recovery area.

3. Make sure the database is in archivelog mode. You check the mode by issuing the following SQL:

   ```
   SQL> select log_mode from v$database;

   LOG_MODE
   ----------
   ARCHIVELOG
   ```

4. The value of the column LOG_MODE is `ARCHIVELOG`, which indicates the database is running in archivelog mode.

5. If the result of the query is different, as in the example shown here, then the database is not running in archivelog mode:

   ```
   SQL> select log_mode from v$database;

   LOG_MODE
   ------------
   NOARCHIVELOG
   ```

6. To enable archivelog mode, follow these steps:

   a. Shut down the database by issuing the following SQL statement:

      ```
      SQL> shutdown immediate
      ```

   b. Start the database in mount mode by issuing the following SQL statement:

      ```
      SQL> startup mount
      ```

c. Enable archivelog mode by issuing the following command:

```
SQL> alter database archivelog;
```

d. At this point, you can open the database for business, but since your objective is to enable flashback, go to the next step.

7. Make sure the database is in either mounted or open state by issuing the following SQL statement:

```
SQL> select OPEN_MODE from v$database;

OPEN_MODE
---------
MOUNTED
```

8. If the database is not even mounted, then mount it:

```
SQL> alter database mount;

Database mounted.
```

The final line confirms that the database is now mounted.

■ **Note** In some earlier versions of the Oracle Database, you could enable flashback mode only when the database is mounted, not open.

---

9. Enable flashback for the database by issuing the following SQL statement:

```
SQL> alter database flashback on;

Database altered.
```

The database is now in flashback mode. You can open the database now (if not open already).

How It Works

When the database is in flashback mode, it generates flashback logs as a result of changes to the data. These flashback logs are later used to roll the database to a previous state. However, the flashback logs capture only the changes to the data blocks, which may not be enough for rebuilding a consistent database. In addition to the flashback logs, the rollback process needs archived logs. Therefore, the database must also be in archivelog mode to enable flashback.

The flashback logs are stored in the fast recovery area, which is described in Chapter 3. The fast recovery area is the only place the flashback logs can be stored. Therefore, it's necessary to enable a fast recovery area with the appropriate size to enable flashback in the database. You can learn how to size the fast recovery area in Recipe 3-16. One of the inputs to the calculations is the estimated size of the total flashback logs generated. You will learn how to estimate that value in Recipe 13-8.

13-3. Disabling Flashback on a Database

Problem

You want to disable flashback mode for a database.

Solution

Disable flashback mode by issuing the following SQL statement:

```
SQL> alter database flashback off;
Database altered.
```

Now the database is running in nonflashback mode.

How It Works

When the database is taken out of flashback mode, the flashback logs are not generated anymore. You can check that the database has indeed been taken off flashback mode by issuing the following query:

```
SQL> select flashback_on from v$database;

FLASHBACK_ON
------------
NO
```

The result shows NO, confirming that the database is not running in flashback mode now.

13-4. Flashing Back a Database from RMAN

Problem

You want to flash a database back to a point in time in the past through RMAN.

Solution

When you want to flash the database back to a time in the past, you have a few choices in deciding when to flash back to. You can flash back to the following:

- A specific point in time, specified by date and time
- A specific SCN number
- The last `resetlogs` operation
- A named restore point

We describe each of these scenarios in the following sections. Each of the solutions, however, has some common tasks before and after the actual flashback.

Common Presteps

The following are the "common presteps" to follow for any type of full database flashback procedure:

1. Check how far back into the past you can flash back to. Refer to Recipe 13-6.
2. Connect to RMAN:

   ```
   rman target=/
   ```

3. Shut the database down:

   ```
   RMAN> shutdown immediate
   ```

4. Start the database in mount mode:

   ```
   RMAN> startup mount
   ```

This completes the preflashback steps.

Common Poststep

After the flashback operation, you will open the database with the clause `resetlogs`, as shown in the following actions in RMAN:

```
RMAN> alter database open resetlogs;

database opened
```

It's important to open the database in resetlogs mode since the flashback operation performs a point-in-time recovery, which is a form of incomplete recovery. For more information about incomplete recovery, refer to Chapter 12.

Solution 1: Flashing Back to a Specific SCN

In this example, you will see how to flash back a database to a specific SCN, which is the most precise flashback procedure possible. Here are the steps to follow:

1. First, check the SCN of the database now. Connecting as `sys` or any other DBA account, issue the following SQL statement:

   ```
   SQL> select current_scn
     2  from v$database;

   CURRENT_SCN
   -----------
       1137633
   ```

   The output shows the current SCN is 1,137,633. You can flash back to an SCN prior to this number only.

2. Execute the "common presteps" 1 through 4.

3. Flash the database back to your desired SCN. For instance, to flash back to SCN 1,050,951, issue the following RMAN command:

```
RMAN> flashback database to scn 1050951;

Starting flashback at 03-AUG-12
using target database control file instead of recovery catalog
allocated channel: ORA_DISK_1
channel ORA_DISK_1: SID=16 device type=DISK

starting media recovery
media recovery complete, elapsed time: 00:00:01

Finished flashback at 03-AUG-12
```

   This command flashed the database back to the desired SCN.

4. You can open the database now for regular operations by executing the "common poststep."

5. However, you may not be certain whether you have flashed back to the exact point in time you wanted to be at. To determine whether you have, you can open the database in read-only mode:

```
RMAN> alter database open read only;

Database opened.
```

6. Check the data in the table. For instance, the purpose of the flashback was to undo the changes done to the interest calculation table, so you can check the interest table to see whether the values are 0.

7. If you have not gone far back into the past, you can start the flashback process again to flash back to a different SCN. Start with step 2—shut down the database, start up in mount mode, and then flash back.

```
RMAN> flashback database to scn 2981100;
```

   Note that you can use a SCN after the SCN you flashed back to earlier as shown here in this example. Obviously however, you can't flash back to an SCAN more than the current one; that will be akin to flashing back to the future.

8. Once again, open the database in read-only mode, and check the data to make sure you are at a point you want to be. If you are not there, you can redo the steps.

9. Once you are satisfied you have arrived at a point where you want to be, follow step 2 of the "common poststeps" to open the database for regular operation.

Now the database is at the point in time in the past you want to be.

Solution 2: Flashing Back to a Specific Time

You want to flash the database to a specific time, not an SCN. Here are the steps to follow:

1. Follow "common presteps" 1 through 4.
2. Use the following command to flash back to a time just two minutes ago. Since a day has 24 hours, with 60 minutes each, 2 minutes happen to be 2/60/24 of a day:

```
RMAN> flashback database to time 'sysdate-2/60/24';

Starting flashback at 03-AUG-12
using channel ORA_DISK_1

starting media recovery

archived log for thread 1 with sequence 70 is already on disk as
file +FRA/cdb1/archivelog/2012_07_22/thread_1_seq_70.279.789267613
archived log for thread 1 with sequence 71 is already on disk as
file +FRA/cdb1/archivelog/2012_07_22/thread_1_seq_71.280.789285641
media recovery complete, elapsed time: 00:00:40
Finished flashback at 03-AUG-12
```

3. If you want to flash back to a specific time, not in reference to a time such as sysdate, you can use the timestamp instead of a formula:

```
RMAN> flashback database to time "to_date('08/03/2012 22:00:00','mm/dd/yyyyhh24:mi:ss')";
```

   This flashes the database back to that specific timestamp.

4. As in the first solution, you can open the database in read-only mode at this time to check whether you have traversed far enough into the past.
5. If you haven't, you can start the process once again—shut down immediately, start in mount mode, flash back to a different time, and then open the database in read-only mode.
6. Once you are satisfied you have arrived at the desired point in time, shut the database down and follow the "common poststep" to open the database for regular operation.

The database is now as of August 3rd, 2012 at 10:00:00 p.m.

Solution 3: Flashing Back to a Restore Point

In this solution, you will learn how to flash back the database to a restore point. You can learn about creating restore points in Recipe 13-9 and Recipe 13-10. Here are the steps to follow to flash back to a restore point:

1. Follow "common presteps" 1 through 4.
2. To flash back to a restore point named grp6, issue the following SQL:

```
RMAN> flashback database to restore point grp6;

Starting flashback at 03-AUG-12
using channel ORA_DISK_1
```

```
starting media recovery

archived log for thread 1 with sequence 70 is already on disk as
file +FRA/cdb1/archivelog/2012_07_22/thread_1_seq_70.279.789267613
media recovery complete, elapsed time: 00:00:01
Finished flashback at 03-AUG-12
```

3. At this time you can open the database in read-only mode and check the data, as described in the first solution of this recipe. If the flashback is not far enough in the past, or too far, you can flash back to another restore point—grp5, for instance. In that case, you repeat the steps: shut down, start, and flash back. To flash back to restore point grp5, you issue the following RMAN command:

```
RMAN> flashback database to restore point grp5;
```

4. Execute the "common poststep" to open the database for normal operation.

The database is now as of the time when the restore point rp5 was created.

Solution 4: Flashing Back to Before the Last resetlogs Operation

You have opened the database with the `resetlogs` clause, and that was probably a mistake. Now you want to revert the changes to the last `resetlogs` operation. Here are the steps to accomplish that:

1. Execute the "common presteps" 1 through 4.
2. Use the following command to flash back the database to the last `resetlog` operation:

```
RMAN> flashback database to before resetlogs;

Starting flashback at 03-AUG-12
allocated channel: ORA_DISK_1
channel ORA_DISK_1: SID=17 device type=DISK

starting media recovery
media recovery complete, elapsed time: 00:00:01

Finished flashback at 03-AUG-12
```

3. The database has now been flashed back to the last restore point. Execute the "common poststep" to open the database for normal operation.

How It Works

When the database is in flashback mode, it generates special log files called flashback logs that can be used to flash back the database to a prior point in time. The flashback logs carry the SCN, allowing you to use the SCN as a measuring point to which to flash back. But SCNs are akin to the internal clock of the database, and they also relate to the wall clock. Therefore, when you issue the commands to flash back to a specific timestamp, Oracle automatically determines the SCN associated with the timestamp and rolls back to that SCN.

Similarly, restore points are merely pointers to specific SCNs, so when you flash back to a specific restore point, the database actually issues a flashback to the SCN associated with that restore point. Finally, the database records the SCN when the database was opened with resetlogs; so, again, your flashback command to the last resetlogs operation is merely the same as issuing the flashback to that SCN. You can check the SCN during the last resetlogs operation by issuing the following query:

```
SQL> select resetlogs_change#
  2   from v$database;

RESETLOGS_CHANGE#
-----------------
1070142
```

Flashback does not work in only one direction; it works both back and forth from a point. Of course, you can't go to a point in time in the future, and you can go only as far back into the past as the flashback logs are available. Figure 13-1 shows how the flashback works in both forward and reverse directions from a point.

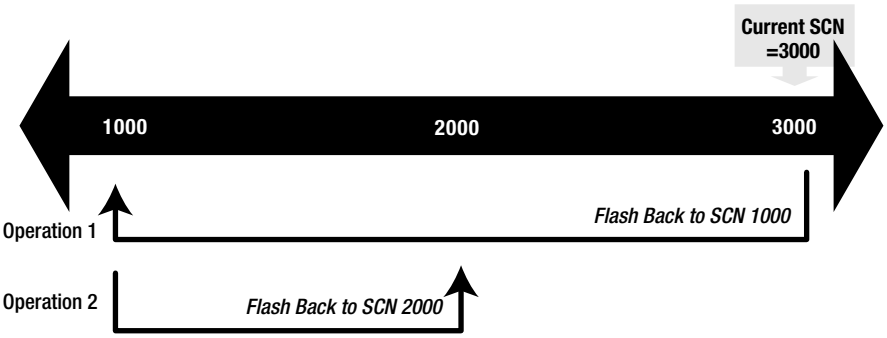

Figure 13-1. *Flashback operations*

Note that Operation 1 flashed the database from the current SCN (3,000) to SCN 1,000. After that was done, before the database opened for read/write access, Operation 2 flashed the database back from SCN 1,000 to SCN 2,000, which is akin to rollforward operations, but we still call it flashback. You can do this operation up to any SCN less than 3,000 any number of times to get to the precise position in time. The lower limit of SCN you can flash back to depends on how much flashback log data is available in the fast recovery area.

■ **Note** To guarantee the ability to flash back to a point in time, you can create guaranteed restore points, discussed in Recipe 13-10.

13-5. Flashing Back a Database from SQL

Problem

You want to flash the database back to a point in time in the past by using SQL statements, not RMAN.

Solution

Like the RMAN approach, several options are available to you in deciding on a reference point to flash back to. You can flash back to the following:

- A specific point in time, specified by date and time
- A specific SCN
- A named restore point

Common Presteps

We'll describe each option's solution in the following sections. All the solutions have some common steps, just like the RMAN approach described in Recipe 13-4. Here are those common tasks:

1. Check how far back into the past you can flash back to. Refer to Recipe 13-6.
2. Connect as a `sysdba` user, and shut down the database:

   ```
   SQL> shutdown immediate
   ```

3. Start the database in mount mode:

   ```
   SQL> startup mount
   ```

This completes the preflashback steps.

Common Poststep

After the flashback operation, you will open the database with the clause `resetlogs`:

```
SQL> alter database open resetlogs;

database opened
```

It's important to open the database in resetlogs mode since the flashback operation performs a point-in-time recovery, which is a form of incomplete recovery. For more information on incomplete recovery, refer to Chapter 12.

Solution 1: Flashing Back to a Time

You have a specific time—such as August 3, 2012, at 10 p.m.—that you want to flash back to. This time must be in the past. Here are the steps to follow:

1. Perform the "common presteps" 1 through 3.
2. Flash the database to your desired timestamp by issuing the following SQL statement:

   ```
   SQL> flashback database to timestamp
     2> to_date('08/03/2012 22:00:00','mm/dd/yyyy hh24:mi:ss');

   Flashback complete.
   ```

   The message "Flashback complete" confirms that the database has been flashed back.
3. As described in the RMAN approach, you can open the database now for regular operations by executing the "common poststep."
4. However, you may not be certain that you have flashed back to the exact point in time at which you wanted to be. To determine whether you have, you can open the database in read-only mode:

   ```
   SQL> alter database open read only;

   Database opened.
   ```

5. Check the data in the tables so you can figure out whether you have flashed back enough in the past or you need to go even further. For instance, the purpose of the flashback was to undo the changes to the interest calculation table, so you can check the interest table to see whether the values are 0.
6. If you have not gone far back into the past, you can start the flashback process again to flash back to a different timestamp. Start with step 2, and execute the "common presteps" and flashback:

   ```
   SQL> flashback database to timestamp
     2> to_date('08/03/2012 21:00:00','mm/dd/yyyy hh24:mi:ss');
   ```

7. Again, open the database in read-only mode, check the data to make sure you are at the point at which you want to be. If you are not there, you can reexecute step 2 through step 6.
8. Once you are satisfied that you have arrived at a point where you want to be, follow the "common poststep" to open the database for regular operation.

The flashback to the timestamp is now complete.

Solution 2: Flashing Back to a Specific SCN

You have a specific SCN to flash back to. This SCN must be less than the current SCN. The steps are the same as for the first solution, except for step 6, in which you substitute the SCN with the timestamp:

1. Find out the current SCN by issuing this query:

   ```
   sql> select current_scn from v$database;

   CURRENT_SCN
   -----------
   1044916
   ```

   From the output, you know that the current SCN is 1,044,916. You can flash back only to a SCN less than this number. These are the steps to flash back to the SCN 1,000,000.

2. Follow the "common presteps" 1 through 3.

3. Issue the following SQL statement to flash back to SCN 1,000,000:

   ```
   SQL> flashback database to scn 1000000;

   Flashback complete.
   ```

4. After the flashback is complete, you can open the database in read-only mode to check the contents.

   ```
   SQL> alter database open read only;

   Database altered.
   ```

5. After the database is opened, you can check the data and determine whether the flashback was done to a time far back enough. If not, you can flash it back once more by repeating the steps: shut down, start up, flash back, and open as read-only.

6. When you want the database to be at a certain point in time, follow the "common poststep" to open the database for normal use.

The database is now flashed back and ready for use.

Solution 3: Restoring to a Restore Point

You can also use the flashback feature to roll a database back to a named restore point. See Recipe 13-9 to learn how to create a restore point. Then use the following steps to revert to such a restore point:

1. Follow the "common presteps."

2. Issue the following SQL statement to flash the database back to, in this example, restore point rp1:

   ```
   SQL> flashback database to restore point rp1;

   Flashback complete.
   ```

3. Similar to the second solution, you can open the database in read-only mode to check whether you have flashed back to a correct place in time:

```
SQL> alter database open read only;

Database altered.
```

4. After the database is opened, you can check the data and determine whether the flashback was done to a time far back enough. If not, you can flash it back once more by repeating the steps: shut down, start up in mount mode, flash back, and open as read-only.
5. When you want the database to be at a certain point in time, follow the "common poststep" to open the database for normal use.

The database is now flashed back and ready for use.

How It Works

The SQL approach works exactly like the RMAN approach described in Recipe 13-4. Refer to the "How It Works" section of that recipe for details.

13-6. Finding Out How Far Back into the Past You Can Flash Back

Problem

You want to flash back the database, and you want to find out how far into the past you can go.

Solution

Query the V$FLASHBACK_DATABASE_LOG view to find out how far into the past you can flash back. For example:

```
SQL> select * from v$flashback_database_log;

OLDEST_FLASHBACK_SCN OLDEST_FL RETENTION_TARGET FLASHBACK_SIZE
-------------------- --------- ---------------- --------------
ESTIMATED_FLASHBACK_SIZE     CON_ID
------------------------ ----------
             2193903 21-JUL-12             1440      367001600
                20570112          0
```

The value of the column OLDEST_FLASHBACK_SCN is 2193903, which indicates you can flash back to the SCN up to that number only, not before that.

The column OLDEST_FLASHBACK_TIME shows the earliest time you can flash back to when you use the timestamp approach shown in Recipe 13-4 and Recipe 13-5. The default display format of a datetime column is just a date, and it does not yield enough information. To see the exact time, you issue the following SQL statement:

```
SQL> select to_char(oldest_flashback_time,'mm/dd/yy hh24:mi:ss')
  2  from v$flashback_database_log;

TO_CHAR(OLDEST_FL)
-----------------
07/21/12 17:39:12
```

The output shows that you can flash back to at most July 21, 2012, at 5:39:12 p.m. when using the timestamp option.

How It Works

You can flash back the database to any point in the past as long as the required flashback logs are available and as long as the required archived logs are available. The archived logs can be either online or on backup, but they must be available.

Information on flashback logs is available on the data dictionary view V$FLASHBACK_DATABASE_LOG.

The dynamic performance view V$FLASHBACK_DATABASE_LOG shows some of the information on flashback operations. Table 13-1 describes the columns of this view.

Table 13-1. *Columns of V$FLASHBACK_DATABASE_LOG*

Column Name	Description
OLDEST_FLASHBACK_SCN	The minimum SCN to which you can flash back the database.
OLDEST_FLASHBACK_TIME	The earliest time to which you can flash back the database.
RETENTION_TARGET	The initialization parameter `db_flashback_retention_target` determines how long the flashback logs are retained, in minutes. The same parameter is shown in this column. See the note after this table for more information.
FLASHBACK_SIZE	The size of flashback logs as of now.
ESTIMATED_FLASHBACK_SIZE	This column is a bit more interesting and explained in detail after this table.
CON_ID	The container ID, in case of a pluggable database (only in case of Oracle Database 12.1 and above). If you don't use a pluggable database, this column will show the default container ID of 0.

To find the value of the retention target set in the database, you can also issue this SQL:

```
SQL> show parameter db_flashback_retention_target
NAME                           TYPE     VALUE
------------------------------ -------- ------
db_flashback_retention_target integer   1440
```

Note that the database initialization parameter `db_flashback_retention_target` sets the target for the flashback operation. Since this is set to 1440 in the solution example, the database tries to keep the logs for 1,440 minutes. The important word here is "tries," not "guarantees." The actual number of logs kept depends on the size of the fast recovery area, which is determined by another database initialization parameter: `db_recovery_file_dest_size`. When the flashback logs fill up the fast recovery area, the database removes the oldest logs to make room for the new ones. The age of the oldest logs removed may potentially be less than 1,440 minutes, which is why 1,440 minutes is merely a target, not a guaranteed value of retention.

So, if the database were to retain the flashback logs for the entire 1,440 minutes, what would the combined size of those flashback logs be?

The column ESTIMATED_FLASHBACK_SIZE answers the question. In the example shown here, the value of this column is 20,570,112, or about 20 MB, while the column FLASHBACK_SIZE is 367,001,600, or about 366 MB, much more than the estimated size. This occurred since the fast recovery area has plenty of space and the older flashback logs are still retained in the fast recovery area. Normally, on a small fast recovery area and very active database, this output is reversed—the estimated size is more than the actual size.

13-7. Estimating the Amount of Flashback Logs Generated at Various Times

Problem

You want to find out how much space the flashback logs are expected to consume in the database at various points in time.

Solution

The solution is rather simple. The Oracle database already has a view that shows the estimated database changes and flashback changes in a one-hour period. This view is V$FLASHBACK_DATABASE_STAT. Here is a sample of how to use the view to identify how much flashback and database change data are generated in hour-long intervals:

```
SQL> alter session set nls_date_format = 'mm/dd/yy hh24:mi:ss';

Session altered.

SQL>    select * from v$flashback_database_stat
2       order by begin_time
3       /

BEGIN_TIME        END_TIME          FLASHBACK_DATA    DB_DATA  REDO_DATA
----------------- ----------------- -------------- ---------- ----------
ESTIMATED_FLASHBACK_SIZE     CON_ID
------------------------ ----------
08/03/12 23:09:29 08/04/12 00:53:34           8192    1392640          0
                       0          0
... and so on ...
```

The data of interest is the column ESTIMATED_FLASHBACK_SIZE, which shows the expected flashback log generated in the time period shown by the columns BEGIN_TIME and END_TIME. Using this view, you can see an hour-by-hour progress of the flashback data generation. Issue the following query to find out the estimated total size of the flashback logs at the end of each period:

```
SQL> select end_time, estimated_flashback_size
  2  from v$flashback_database_stat
  3  order by 1
  4  /
```

Here is the output:

```
END_TIME             ESTIMATED_FLASHBACK_SIZE
---------------- --------------------------
08/02/12 19:58:00  73786720
08/02/12 20:53:10 164890123
08/02/12 21:57:37 287563456
... and so on ...
```

Studying the output, you can see the demand for flashback logs went up at 21:57 to 287,563,456, or about 287MB. If you estimate the total size of flashback logs as 190MB, then the older logs will be deleted to make room for the new ones at 21:57. This information helps you when deciding the optimal value of the flashback logs.

How It Works

This view V$FLASHBACK_DATABASE_STAT shows the estimated flashback data within hour-long intervals. Table 13-2 describes the columns of the view.

Table 13-2. *Columns of V$FLASHBACK_DATABASE_STAT*

Column Name	Description
BEGIN_TIME	The beginning of the interval
END_TIME	The end time of the interval
FLASHBACK_DATA	The amount of flashback data generated in bytes in this time interval
DB_DATA	The amount of database change data generated in bytes in this time interval
REDO_DATA	The amount of redo generated in bytes in this time interval
ESTIMATED_FLASHBACK_SIZE	The estimated size of the total flashback logs retained to satisfy the retention target at the end of this time interval, shown in the column END_TIME
CON_ID	The container ID (in case of Oracle Database 12.1 and above)

13-8. Estimating the Space Occupied by Flashback Logs in the Fast Recovery Area

Problem

You want to estimate how much space will be needed for the flashback logs to be retained enough to flash back by a time period specified by the retention target.

Solution

To estimate the total size of all flashback logs required for the retention target, follow these steps:

1. Check the dynamic performance view V$FLASHBACK_DATABASE_LOG:

```
SQL> select * from v$flashback_database_log;

OLDEST_FLASHBACK_SCN OLDEST_FL RETENTION_TARGET FLASHBACK_SIZE
-------------------- --------- ---------------- --------------
ESTIMATED_FLASHBACK_SIZE     CON_ID
------------------------ ----------
             2193903 21-JUL-12             1440      367001600
                21479424          0
```

2. Note the value of ESTIMATED_FLASHBACK_SIZE, which is 21,479,424, or about 20MB in this case. This should ideally be your size of the flashback logs.

How It Works

It is not necessary for the database to hold on to the flashback logs. If the space inside the flashback recovery area is under pressure, Oracle automatically deletes the oldest flashback logs to make room for the new ones. Even though the retention target is set, there is no guarantee that Oracle can actually flash back to that point in the past. Since flashback logs are removed only when there is no space, if you size the flashback recovery area large enough, no flashback logs that are required to flash the database back by the retention target need to be deleted. In this recipe, you have identified how many flashback logs would need to be retained to meet the retention target requirement.

13-9. Creating Normal Restore Points

Problem

You want to create normal (or nonguaranteed) restore points that you can later flash back to.

Solution

Execute a statement such as the following, which creates a restore point named rp1:

```
SQL> create restore point rp1;

Restore point created.
```

The restore point is now created. You can flash back to the rp1 restore point later, as explained in Recipe 13-4 and Recipe 13-5.

How It Works

Restore points are named positions in time. While flashing a database back, you can specify a restore point as a destination instead of specifying an SCN or timestamp. However, flashing back to a restore point is possible only if the flashback logs are available for the time associated with the restore point. Because the restore points created by following this recipe are not guaranteed, they are known as unguaranteed or normal restore points. Normal restore points are the default type.

13-10. Creating Guaranteed Restore Points

Problem

You want to create guaranteed restore points to ensure that you can flash back to them as needed. You want to require the database to retain any needed logs to support those points.

Solution

Add the `guarantee` keyword to your `create restore point` command. For example:

```
SQL> create restore point rp2 guarantee flashback database;

Restore point created.
```

Restore point rp2 is now created as a guaranteed restore point.

How It Works

For a description of restore points and how they work, refer to the "How It Works" section of Recipe 13-9. As described in that recipe, merely defining a restore point does not mean you can flash back to the associated point in time. Flashback logs are deleted by the database automatically when the space in the fast recovery area is inadequate for an incoming backup. It's entirely possible then for the logs required by a given restore point to be deleted, making that restore point useless.

If you try to flash the database to a point for which no flashback logs are available, you will see the following error message:

```
ORA-38729: Not enough flashback database log data to do FLASHBACK.
```

This message means the database does not have the flashback logs needed to go back to the restore point (or time or SCN) that you've specified. By adding the word `guarantee` to your `create restore point` command, you prevent the database from deleting any needed logs for whatever restore point you are creating.

If you have a guaranteed restore point, you can't change the log mode of the database to noarchivelog. Here is the error you will get:

```
SQL> alter database noarchivelog;
alter database noarchivelog
*
ERROR at line 1:
ORA-38781: cannot disable media recovery - have guaranteed restore points
```

This is due to fact that guaranteed restore points allow you to flash the database to that point in time and to accomplish that the Oracle Database must read and access the archived logs.

■ **Caution** When a guaranteed restore point is defined, the associated flashback logs are never deleted unless the restore point is dropped. This will reduce the available space in the fast recovery area. A filled-up fast recovery area will cause the database instance to abort, with the failure in the recovery writer (RVWR) process. So, create guaranteed restore points only when you need to go back to them after some preestablished event to be completed in the near future, such as doing a test run of the application and then reverting to the starting data sets. After the test is completed, drop the guaranteed restore points.

13-11. Listing Restore Points

Problem

You want to list the various restore points in the database and the information about them.

Solution

Query the view V$RESTORE_POINT. For example:

```
SQL> col time format a32
SQL> col name format a10
```

```
SQL> select name, DATABASE_INCARNATION#, SCN, time
  2  from v$restore_point
  3  order by scn;

NAME       DATABASE_INCARNATION#        SCN TIME
---------- -------------------- ---------- -------------------------------
RP1                           2    2193924 21-JUL-12 05.39.31.000000000 PM
GRP1                          2    2194050 21-JUL-12 05.44.48.000000000 PM
GRP2                          2    2194061 21-JUL-12 05.44.58.000000000 PM
RP2                           2    2194077 21-JUL-12 05.45.16.000000000 PM
GRP3                          2    2195811 21-JUL-12 06.19.15.000000000 PM
GRP4                          2    2195894 21-JUL-12 06.19.47.000000000 PM
GRP5                          2    2196549 21-JUL-12 06.39.20.000000000 PM
GRP6                          2    2196630 21-JUL-12 06.40.30.000000000 PM
GRP7                          3    2197224 03-AUG-12 11.04.38.000000000 PM
GRP8                          3    2197263 03-AUG-12 11.05.33.000000000 PM
RP9                           3    2197281 03-AUG-12 11.05.50.000000000 PM
GRP11                         4    2199532 04-AUG-12 01.34.37.000000000 AM
... and so on ...
```

The various columns of the view are described in the "How It Works" section.

How It Works

When you create a restore point as guaranteed, the database marks the flashback logs as not to be removed when the fast recovery area runs out of space. The space occupied by these specially marked flashback logs is shown under the column STORAGE_SIZE in the view V$RESTORE_POINT.

Table 13-3 describes the columns of the view V$RESTORE_POINT.

Table 13-3. *Columns of V$RESTORE_POINT*

Column Name	Description
SCN	This is the SCN of the database when the restore point was created.
DATABASE_INCARNATION#	This column displays the incarnation of the database when this restore point was created. If the database was flashed back and then opened with `resetlogs`, it creates a new incarnation of the database.
GUARANTEE_FLASHBACK_DATABASE	If the restore point is a guaranteed one, this column holds the value `YES`.
STORAGE_SIZE	This is the storage occupied by the flashback logs of the guaranteed restore points. In case of normal restore points, this value is 0.
TIME	This is the timestamp when the restore point was created.
NAME	This is the name of the restore point.
PRESERVED	This is a new column in Oracle Database 11g. It shows whether the restore point must be explicitly deleted.
RESTORE_POINT_TIME	This shows whether you specified a specific time when the restore point was supposed to be taken. If you didn't specify a time, it's NULL.
CON_ID	The container ID in case of Oracle Database 12.1 and above.

13-12. Dropping Restore Points

Problem

You want to drop a specific restore point.

Solution

To drop a restore point named rp2, whether normal or guaranteed, simply execute the following SQL statement:

```
SQL> drop restore point rp2;

Restore point dropped.
```

To list the restore points defined in the database, use Recipe 13-11.

How It Works

Normal restore points are merely pointers to the SCNs at the time they were defined. They do not consume any space. Guaranteed restore points mark the flashback logs necessary to enable flashback to a specific point in time, and those flashback logs do take up space. When you drop a guaranteed restore point, you will see an immediate increase in the available space in the fast recovery area. To check the available space in the fast recovery area, refer to Recipe 3-4.

13-13. Recovering a Dropped Table

Problem

You accidentally dropped a table that should not have been dropped. You want to reinstate the table without doing a database recovery.

Solution

If you dropped the table just moments ago, it is not actually dropped; it is placed in the recycle bin. Assume that you dropped the table ACCOUNTS and want to revive it. You can resurrect that table from the recycle bin by following these steps:

1. Log on to the database as the table owner.
2. Check whether the table exists in the recycle bin. Issue the SQL*Plus command show recyclebin:

```
SQL> show recyclebin

SQL> show recyclebin
ORIGINAL NAME    RECYCLEBIN NAME                OBJECT TYPE  DROP TIME
---------------- ------------------------------ ------------ -------------------
TEST             BIN$xmtCqONCcZjgQ4CohAoD7Q==$0 TABLE        2012-08-04:01:53:19
```

The presence of the table TEST under the column ORIGINAL_NAME indicates that the table is still present in the recycle bin and can be revived. If you see multiple entries with the same ORIGINAL_NAME, it indicates the table was dropped, another table was created with the same name, that table was dropped, too, and so on, for however many duplicate entries you have. Recipe 13-14 shows how to handle a situation in which you have duplicate names in the recycle bin.

3. Revive the table from the recycle bin by issuing the following SQL*Plus command:

```
SQL> flashback table test to before drop;

Flashback complete.
```

The table is now available in the database.

How It Works

In Oracle Database 10g, when you drop a table, the table is not really dropped. Rather, the table is renamed. For instance, in the example in this recipe, when the table TEST was dropped, the table was actually renamed to BIN$xmtCqONCcZjgQ4CohAoD7Q==$0. That name is cryptic enough that it would never be used and thus would never conflict with a real name by any user. Since the table is merely renamed and not dropped, the data in the table is still available. When you issue the flashback command in step 2, Oracle Database starting with 10g merely renames the table to the original name. However, the dropped table does not show up in the data dictionary views USER_TABLES and ALL_TABLES.

```
SQL> select table_name
  2  from user_tables;

no rows selected
```

However, the view TAB shows this renamed table:

```
SQL> select * from tab;

TNAME                          TABTYPE CLUSTERID
------------------------------ ------- ---------
BIN$xmtCqONCcZjgQ4CohAoD7Q==$0 TABLE
```

If you check the USER_SEGMENTS dictionary view, the segments will be there:

```
SQL> col segment_name format a30
SQL> select segment_type, segment_name
  2  from user_segments;
SEGMENT_TYPE    SEGMENT_NAME
-------------------------------------
TABLE   BIN$xmtCqONDcZjgQ4CohAoD7Q==$0
INDEX   BIN$FPl4bnVgTH2ZIr1uc31OHg==$1
INDEX   BIN$c7f1XmKBQjiVXy2j/NcJqA==$1
```

The indexes are those of the table. When the table was dropped, the indexes were not dropped. They were renamed, just like the table.

If you make a mistake in identifying the correct table, Oracle Database returns an ORA-38305 error, as shown in the following example where you are trying to revive a table named ACCOUNTS that does not exist in the recycle bin:

```
SQL> flashback table accounts to before drop;
flashback table accounts to before drop
*
ERROR at line 1:
ORA-38305: object not in RECYCLE BIN
```

The error says it all.

■ **Tip** If you want to delete a table permanently, without sending it to the recycle bin, then use the `purge` clause in the `drop` statement. For example:

```
SQL> drop table test purge;
Table dropped.
```

The table is now completely dropped; similar to the pre-10g behavior, it does not go to the recycle bin.

13-14. Undropping a Table When Another Exists with the Same Name

Problem

You had a table called ACCOUNTS that was dropped, and since then you created another table also called ACCOUNTS. Now you want to reinstate the first table ACCOUNTS from the recycle bin.

Solution

There are two potential solutions:

- Drop the existing table so there will be no conflict for the name of the table undropped.
- Undrop the table but reinstate it to a different name.

Here are the solutions in detail.

Solution 1: Dropping the Existing Table

The easiest approach is, of course, to drop the existing table. The flashed-back table then comes on the database without any problems.

Solution 2: Renaming the Reinstated Table

The alternative approach is safer because you do not need to drop anything. When you flash back a table to undrop it, you can optionally rename it. In this case, when you flash back the table ACCOUNTS, you want to reinstate it as NEW_ACCOUNTS.

```
SQL> flashback table accounts to before drop rename to new_accounts;

Flashback complete.
```

The existing table still remains as ACCOUNTS, but the reinstated table is renamed to NEW_ACCOUNTS.

How It Works

When you flash back a table from the recycle bin, a table with that name must not already exist in the database. Suppose you are trying to revive a table called ACCOUNTS but it already exists. In that case, the flashback statement returns with an error: ORA-38312:

```
SQL> flashback table accounts to before drop;

Flashback complete.

SQL> flashback table accounts to before drop;
flashback table accounts to before drop
*
ERROR at line 1:
ORA-38312: original name is used by an existing object
```

■ **Note** Suppose there are two tables in the recycle bin with the same name—ACCOUNTS, as shown here:

```
ORIGINAL NAME RECYCLEBIN NAME                OBJECT TYPE  DROP TIME
------------- ------------------------------ ------------ -------------------
ACCOUNTS      BIN$xmtCqONLcZjgQ4CohAoD7Q==$0 TABLE        2012-08-04:02:09:42
ACCOUNTS      BIN$xmtCqONKcZjgQ4CohAoD7Q==$0 TABLE        2012-08-04:02:09:24
TEST1         BIN$xmtCqONDcZjgQ4CohAoD7Q==$0 TABLE        2012-08-04:02:05:45
```

Now you issue this:

```
SQL> flashback table accounts to before drop;
```

Which table ACCOUNTS will be reinstated?

The table that was dropped last will be reinstated; that is, the table that shows up first will be reinstated. Pay attention to this behavior while reinstating a table from the recycle bin.

13-15. Undropping a Specific Table from Two Dropped Tables with the Same Name

Problem

You had a table called ACCOUNTS, which you dropped. Later you created a table, again called ACCOUNTS, and dropped that, too. Now you want to revive the table ACCOUNTS, the one that was dropped first.

Solution

To reinstate a specific dropped table, follow the steps:

1. First find out the presence of these objects in the recycle bin:

```
ORIGINAL NAME RECYCLEBIN NAME                OBJECT TYPE  DROP TIME
------------- ------------------------------ ------------ -------------------
ACCOUNTS      BIN$xmtCqONLcZjgQ4CohAoD7Q==$0 TABLE        2012-08-04:02:09:42
ACCOUNTS      BIN$xmtCqONKcZjgQ4CohAoD7Q==$0 TABLE        2012-08-04:02:09:24
TEST1         BIN$xmtCqONDcZjgQ4CohAoD7Q==$0 TABLE        2012-08-04:02:05:45
```

   Note there are two different tables with the same name—ACCOUNTS.

2. Decide which of the two accounts tables to revive. The column DROP_TIME helps in your decision; it shows when each table was dropped. In your case, you want to recover the one that was dropped earlier. If you issue the statement `flashback table accounts to before drop`, the more recently dropped table will be revived—not what you want in this scenario.

3. To revive the earlier table, the one that was dropped first, issue the `flashback table` command, giving the recycle bin name as the table name:

```
SQL> flashback table "BIN$bQ8QU1bWSD2Rc9uHevUkTw==$0" to before drop;
Flashback complete.
```

   Be sure to put the recycle bin name— `BIN$xmtCqONKcZjgQ4CohAoD7Q==$0`—in double quotes. The double quotes are necessary because of the presence of special characters in the name.

4. Check the recycle bin. You will see only one table now:

```
SQL> show recyclebin
ORIGINAL NAME RECYCLEBIN NAME                OBJECT TYPE  DROP TIME
------------- ------------------------------ ------------ -------------------
ACCOUNTS      BIN$xmtCqONLcZjgQ4CohAoD7Q==$0 TABLE        2012-08-04:02:09:42
```

   There is just one table in the recycle bin. You have successfully restored the earlier version of the table.

How It Works

As mentioned in Recipe 13-13, a `drop table` command in Oracle Database 10g and later does not actually drop a table; it merely renames it to a name with a lot of special characters. An example of such a name is the `"BIN$xmtCqONKcZjgQ4CohAoD7Q==$0"` name shown in this recipe. While reviving a table in the recycle bin, you can use its original name or the special recycle bin name.

In most cases, you can use the original name. Sometimes, though, you can't use the original name. One such example could be when you drop a table and create another with the same name. When you re-create a table that was dropped before and then drop the re-created one, the table name is same—ACCOUNTS, in this example—but the recycle bin names for each of those two tables are unique. To reinstate a specific dropped table, you should specify the recycle bin name instead of the real name.

13-16. Checking the Contents of the Recycle Bin

Problem

You want to see the objects in the recycle bin.

Solution

You can display the objects in your own recycle bin in two ways:

- Use the SQL*Plus command `show recyclebin`:

```
SQL> show recyclebin
ORIGINAL NAME RECYCLEBIN NAME                OBJECT TYPE  DROP TIME
------------- ------------------------------ ------------ -------------------
TEST          BIN$xmtCqONMcZjgQ4CohAoD7Q==$0 TABLE        2012-08-04:02:15:11
TEST1         BIN$xmtCqONDcZjgQ4CohAoD7Q==$0 TABLE        2012-08-04:02:05:45
```

  The command `SHOW RECYCLEBIN` shows some pertinent details for all tables in the recycle bin. However, the command does not show corresponding indexes, triggers, and so on.

- To get information on all objects in the recycle bin, including indexes and triggers, query the view USER_RECYCLEBIN, as shown in the following example:

```
SQL> select * from user_recyclebin;

OBJECT_NAME
--------------------------------------------------------------------------------
ORIGINAL_NAME
--------------------------------------------------------------------------------
OPERATION TYPE                      TS_NAME
--------- ------------------------- ------------------------------
CREATETIME          DROPTIME              DROPSCN
------------------- ------------------- ----------
PARTITION_NAME
--------------------------------------------------------------------------------
CAN CAN    RELATED BASE_OBJECT PURGE_OBJECT      SPACE
--- --- ---------- ----------- ------------ ----------
BIN$xmtCqONDcZjgQ4CohAoD7Q==$0
```

```
TEST1
DROP      TABLE                     SYSAUX
2012-08-04:02:05:30 2012-08-04:02:05:45    2207258

YES YES      90902       90902        90902          8

BIN$xmtCqONMcZjgQ4CohAoD7Q==$0
TEST
DROP      TABLE                     SYSAUX
2012-08-04:02:09:38 2012-08-04:02:15:11    2208461

YES YES      90903       90903        90903          8
```

To check the recycle bin of all users, check the view DBA_RECYCLEBIN:

```
SQL> select * from dba_recyclebin;
```

The columns are the same as user_recyclebin, except the additional column—OWNER—that shows the owner of the dropped object. In Oracle 12.1, a new view CDB_RECYCLEBIN shows the recycle bin for the container database.

How It Works

When a table is dropped in Oracle Database 10g Release 1 and newer, it is not actually dropped. It's merely renamed to a different name, such as BIN$UawCFy69TUyc9DgR50AEMw==$0. A record is placed in the table RECYCLEBIN$ (in the sys schema) for that table. The view USER_RECYCLEBIN is a join between, among other tables, the OBJ$ (the objects in the database) and RECYCLEBIN$ tables in the sys schema.

13-17. Restoring Dependent Objects of an Undropped Table

Problem

You want to recover all the subordinate objects, such as the indexes, constraints, and so on, of a table that has been undropped.

Solution

Here are the steps to restore the dependent objects:

1. First, check the contents of the recycle bin to get an inventory of what is available. This is an important step; do not skip it. The following is the query you want to execute:

```
SQL> col type format a5
SQL> col original_name format a15
SQL> col object_name format a15
SQL> select original_name, object_name, type, can_undrop
  2  from user_recyclebin;

ORIGINAL_NAME   OBJECT_NAME                    TYPE      CAN
--------------- ------------------------------ --------- ---
IN_ACC_01       BIN$xmtCqONUcZjgQ4CohAoD7Q==$0 INDEX     NO
```

```
IN_ACC_02       BIN$xmtCqONVcZjgQ4CohAoD7Q==$0 INDEX     NO
TR_ACC_01       BIN$xmtCqONWcZjgQ4CohAoD7Q==$0 TRIGGER   NO
ACCOUNTS        BIN$xmtCqONXcZjgQ4CohAoD7Q==$0 TABLE     YES
```

The most important column is the column CAN_UNDROP. If this column is YES, then you can undrop an object cleanly without any additional efforts. Objects with CAN_UNDROP = NO can still be reinstated, but you have to change their names to the original names manually.

■ **Tip** The ORIGINAL_NAME column shows the original names. Once a table is undropped, the recycle bin information is removed, and you will never be able to see the original names of the dependent objects, such as triggers and indexes of that table. So, save the output of this query before you go to the next step.

---

2. Now undrop the table ACCOUNTS by executing the following SQL statement:

```
SQL> flashback table accounts to before drop;

Flashback complete.
```

The table is now available in the database.

3. Display the constraints of the table:

```
SQL> select constraint_type, constraint_name
  2  from user_constraints
  3  where table_name = 'ACCOUNTS';

C CONSTRAINT_NAME
- ------------------------------
P BIN$ncFOiaduRZeURXatWq8lyA==$0
C BIN$782qhcPvQbajusPeAEiR3Q==$0
```

The flashback (or the undrop) brought back the primary key and check constraints but not the foreign keys, if there were any. The foreign keys are lost forever.

4. Change the names of the reinstated objects to their original names, if you know what they were. For example, if you know the original name of the constraint BIN$ncFOiaduRZeURXatWq8lyA==$0 was pk_accounts, you can issue the following query to restore the original name:

```
SQL> alter table accounts rename constraint "BIN$ncFOiaduRZeURXatWq8lyA==$0" to
pk_accounts;

Table altered.

SQL> alter table accounts rename constraint "BIN$782qhcPvQbajusPeAEiR3Q==$0" to
ck_acc_01;

Table altered.
```

If you don't have the names, use any human-readable name you consider appropriate.

5. Now check the indexes of the newly reinstated table:

```
SQL> select index_name
  2  from user_indexes
  3  where table_name = 'ACCOUNTS';

INDEX_NAME
------------------------------
BIN$9POlL6gfQK6RBoOK4klc3Q==$0
BIN$PookVi5nRpmhmPaVOThGQQ==$0
BIN$fzY77+GmTzqz/3u4dqac9g==$0
```

6. Note the names, and compare them to the names you got in step 1. It's not easy, but you can make a clear connection. Using the output from step 1, rename the indexes:

```
SQL> alter index "BIN$9POlL6gfQK6RBoOK4klc3Q==$0" rename to IN_ACC_01;
Index altered.

SQL> alter index "BIN$PookVi5nRpmhmPaVOThGQQ==$0" rename to SYS_C005457;
Index altered.

SQL> alter index "BIN$fzY77+GmTzqz/3u4dqac9g==$0" rename to in_acc_02;
Index altered.
```

7. Finally, make sure the indexes are in place and have correct names:

```
SQL>    select index_name
   2    from user_indexes
   3    where table_name = 'ACCOUNTS';

INDEX_NAME
--------------------------------------
IN_ACC_01
SYS_C005457
IN_ACC_02
```

8. Check the triggers on the reinstated table:

```
SQL>  select trigger_name
   2  from user_triggers;

TRIGGER_NAME
------------------------------------------------
BIN$dt6tBSIWSn+F5epvjybKmw==$0
```

9. Rename triggers to their original names:

```
SQL> alter trigger "BIN$dt6tBSIWSn+F5epvjybKmw==$0" rename to tr_acc_01;
Trigger altered.
```

10. Check the triggers now to make sure they are named as they were originally:

```
SQL>    select trigger_name
   2    from user_triggers;

TRIGGER_NAME
------------------------------------
TR_ACC_01
```

This confirms you reinstated all the dependent objects.

How It Works

In Oracle Database 10g Release 1 and newer, when a table is dropped, the table is not actually dropped; it is merely renamed to a system-generated name and marked as being in the recycle bin. Likewise, all the dependent objects of the table—triggers, constraints, indexes—are also not dropped; they are renamed as well and continue to exist on the renamed table. When you flash back the table to before the drop, or undrop the table, these dependent objects are not undropped. But those objects do exist, and you can rename them to their original names. The only exceptions are foreign key constraints, which are lost when a table is dropped.

13-18. Turning Off the Recycle Bin

Problem

You want to turn off the recycle bin behavior; that is, you want behavior like in Oracle9i where a dropped table just gets dropped permanently.

Solution

You can modify the recycle bin behavior so that dropped objects do not go to the recycle bin. Instead, they simply get dropped permanently. The parameter that influences this is `recyclebin`. You can set this parameter at the session level or the system level.

Set the session parameter to disable the recycle bin at the session level:

```
SQL> alter session set recyclebin = off;

Session altered.
```

After setting `recyclebin` to `off`, if you drop a table, the table is completely dropped:

```
SQL> drop table accounts;

Table dropped.
```

Now, if you check the recycle bin:

```
SQL> show recyclebin
```

the command returns no output, indicating that the recycle bin is empty.

You can turn off the recycle bin for the entire database by putting this parameter in the parameter file and restarting the database:

```
recyclebin = off
```

If the recycle bin is turned off at the system level, you can turn it on at the session level, and vice versa.

How It Works

In Oracle Database 10g Release 1 and newer, when the tables are dropped, they are really not dropped. Instead, they are renamed and marked to be in the recycle bin. This is the default behavior. By executing the statement `alter session set recyclebin = off` at the session level, the behavior is changed to the pre-10g one; that is, the table is actually dropped as a result of the `drop` command, not renamed to be placed in the recycle bin.

SHOULD YOU TURN OFF THE RECYCLE BIN?

Even though you can turn off the recycle bin at the system level, in our opinion there is no valid reason to do so. Here are some arguments against recycle bins:

- They take up space, since the dropped objects are not actually dropped.
- They make the free-space calculations erroneous because they are dropped but still counted as occupied space.
- They show up in a user's list of tables, which can be confusing. And the names are confusing.
- In some environments, such as data warehouses, a lot of tables are created and dropped rapidly. Dropping those tables is permanent, and there is never a need to undrop them.

Each of these arguments can be countered, as shown here:

- They take up space, but the space is immediately deallocated and given to the segment that needs it, if there is a space pressure in the tablespace. So, the space is not taken up in a practical sense.
- The free-space calculations exclude the recycle bin objects, so the free space reported is accurate.
- The recycle bin object shows up in TAB but not in the view USER_TABLES. Most scripts are written against USER_TABLES, not against TAB, so this is not a real concern.

The last argument has some merit. Ordinarily, this should not cause any issues, since the recycle bin objects are not counted toward the user's total used space. But if you would rather not see the recycle bin objects, you can turn it off for that session only.

So, as you can see, there is no real reason behind turning off the recycle bin at the system level (or mimicking the 9i behavior). On the other hand, if you disable it, you will lose a valuable feature—a safety net of sorts while dropping tables. So, we strongly recommend against turning off the recycle bin.

13-19. Clearing the Recycle Bin

Problem

You want to remove all dropped objects from the recycle bin.

Solution

You can clean up the recycle bin using the purge statement, which clears the recycle bin of the currently logged-on user. For example:

```
SQL> purge recyclebin;

Recyclebin purged.
```

Each user has a logically individual recycle bin. If you want to clear the recycle bins of all users in the database, you should purge dba_recyclebin, as shown here:

```
SQL> purge dba_recyclebin;

DBA Recyclebin purged.
```

This clears all the data from all the recycle bins.

How It Works

Note from the earlier recipes that when a table is dropped, it's not really dropped. Instead, the table is renamed and marked to be in the recycle bin. The statement purge recyclebin merely drops all the objects that were marked to be in the recycle bin.

CALLING PURGE IN PL/SQL

PURGE is a DDL, not DML, statement. The difference is not significant when you use it in the SQL*Plus command line as shown in the examples, but it is important to understand the difference when writing a PL/SQL routine. You can't call it in PL/SQL code as shown here:

```
SQL> begin
  2     purge recyclebin;
  3  end;
  4  /
   purge recyclebin;
         *
ERROR at line 2:
ORA-06550: line 2, column 10:
PLS-00103: Encountered the symbol "RECYCLEBIN" when expecting one of the
following:
:= . ( @ % ;
The symbol ":=" was substituted for "RECYCLEBIN" to continue.
```

To call `purge` in a PL/SQL code, you will need to call it as a parameter to `execute immediate`, as shown here:

```
SQL> begin
  2     execute immediate 'purge recyclebin';
  3  end;
  4  /
PL/SQL procedure successfully completed.
```

13-20. Querying the History of a Table Row (Flashback Query)

Problem

You want to find how the values of the columns in a row have changed over a period of time.

Solution

To find all the changes to the row for ACCNO 3760 in table ACCOUNTS, issue the following query:

```
SQL>    select
2       acc_status,
3       versions_starttime,
4       versions_startscn,
5       versions_endtime,
6       versions_endscn,
7       versions_xid,
8       versions_operation
9       from accounts
10      versions between scn minvalue and maxvalue
11      where accno = 3760
12      order by 3
13      /
```

The result comes back as follows:

```
A VERSIONS_STARTTIME    VERSIONS_STARTSCN VERSIONS_ENDTIME     VERSIONS_ENDSCN ➥
VERSIONS_XID    V
- -------------------- --------------------------  ----------- -------------------- -----------
---------------------------------------------
---------------- -
A 12-JUL12 04.38.57 PM          1076867 12-JUL12 04.39.03 PM   1076870
02002F00D8010000 U
I 12-JUL12 04.39.03 PM          1076870 12-JUL12 04.39.12 PM   1076874
08001B00DB010000 U
A 12-JUL12 04.39.12 PM          1076874
07002B0068010000 U
A               12-JUL12 04.38.57 PM    1076867
```

The results show how the values of the column ACC_STATUS were changed at different points in time. Note the column VERSIONS_OPERATION, which shows the DML operation that modified the value of the corresponding row. The values are as follows:

I: Insert

U: Update

D: Delete

In the example output, you can see that on July 12, 2012, at 4:38:57 p.m. (the value of the column VERSIONS_STARTTIME), someone updated the value of a row by using an Update operation. The SCN at that time was 1076867. The ACC_STATUS column was changed to A at that time. This value was unchanged until July 12, 2012, at 4:39:03 p.m. (the value of column VERSIONS_ENDTIME).

As shown in the second record of the output, on July 12, 2012, at 4:39:03 p.m. and at SCN 1076870, another update operation updated the ACC_STATUS to I. This is how you read the changes to the table row where ACCNO is 3760.

Note the line where VERSIONS_ENDTIME is null. This indicates the current row, which has not been changed yet.

In addition to the SCN, you can also use `timestamp` as a predicate, as shown here:

```
SQL>    select
2       acc_status,
3       versions_starttime,
4       versions_startscn,
5       versions_endtime,
6       versions_endscn,
7       versions_xid,
8       versions_operation
9       from accounts
10      versions between timestamp minvalue and maxvalue
11      where accno = 3762
12      order by 3;
```

In the previous example, you specified the predicate to get all the available records. Note line 10:

```
versions between timestamp minvalue and maxvalue
```

This predicate indicates the minimum and maximum values of the timestamps available. You can specify exact values for these as well. To get the versions on July 12 between noon and 3 p.m., you need to rewrite the query by modifying line 10 to this:

```
versions between timestamp to_date('7/12/2012 12:00:00', 'mm/dd/yyyy hh24:mi:ss')
and to_date('7/12/2012 15:00:00', 'mm/dd/yyyy hh24:mi:ss')
```

Instead of using timestamps, you can use SCNs, such as between 1000 and 2000, to get the versions of the row. In that case, line 10 becomes this:

```
versions between SCN 1000 and 2000
```

If you don't see any data under the pseudocolumns, the reasons could be one of the following:

- The information has aged out of the undo segments.
- The database was recycled after the changes occurred.

How It Works

When a row is updated, the database records the relevant change details in the database blocks, in addition to some other related details, such as the SCN of when the change occurred, the timestamp, the type of operation that resulted in the change, and so on—a sort of "metadata" about the changes, if you will. This metadata is stored in pseudocolumns and can be queried afterward.

Table 13-4 describes the flashback query pseudocolumns. The pseudocolumns that start with VERSIONS, such as VERSIONS_STARTTIME, are not actually part of the table. They are computed and shown to the user at runtime. A good everyday example of such a pseudocolumn is ROWNUM, which denotes the serial number of a row in the returned result set. This column is not stored in the table but is computed and returned to the user when the query is executed. Since these columns are not part of the table's definition, they are called pseudocolumns.

Table 13-4. *Flashback Query Pseudocolumns*

Pseudo Column Name	Description
VERSIONS_STARTTIME	This is the timestamp when this version of the row became effective. This is the commit time after the row was changed.
VERSIONS_STARTSCN	This is the SCN when this version became effective.
VERSIONS_ENDTIME	This is the timestamp when the version became old, replaced by a new version. This is the time of commit after the row was changed.
VERSIONS_ENDSCN	This is the SCN when the row's version was changed.
VERSIONS_XID	This is the transaction ID that changed the row's version. This can be joined with the XID column of the dictionary view FLASHBACK_TRANSACTION_QUERY to show the transaction that made this change. The view FLASHBACK_TRANSACTION_QUERY also shows other relevant details of the transaction, such as who did it, when, and so on.
VERSIONS_OPERATION	This is the abbreviated activity code—I, U, or D—for Insert, Update, or Delete that resulted in this version of the row.

13-21. Flashing Back a Specific Table

Problem

You want to flash back a specific table, not the entire database, to a point in time in the past.

Solution

The table can be flashed back with a specialized adaptation of flashback queries. Here are the steps on how to do it:

1. Make sure the table has row movement enabled:

```
SQL>    select row_movement
2    from user_tables
3    where table_name = 'ACCOUNTS';

ROW_MOVE
--------
ENABLED
```

2. If the output comes back as DISABLED, enable it by issuing this SQL statement:

```
SQL> alter table accounts enable row movement;

Table altered.
```

   This prepares the table for flashback.

3. Check the table to see how far into the past you can flash it back. Use Recipe 13-6 to determine how far back into the past you can go.

4. Flash the table back to a specific timestamp:

```
SQL> flashback table accounts to timestamp to_date ('12-JUL-12 18.23.00', 'dd-MON-YY hh24.mi.ss');

Flashback complete.
```

   You can flash back to a specific SCN as well:

5. Check the data in the table to make sure you have flashed back to the exact point you want. If the flashback was not enough, you can flash the table back once more to a point even further in the past. For instance, the previous step reinstated the table as of 6:23 p.m., which was not enough. In this step, you will flash it back to one more minute in the past—to 6:22 p.m.

```
SQL> flashback table accounts to timestamp to_date ('12-JUL-12 18.22.00',
'dd-MON-YY hh24.mi.ss');
Flashback complete.
```

6. If you have gone too far into the past, you can flash "forward" using the same flashback statement:

```
SQL> flashback table accounts to timestamp to_date ('12-JUL-12 18.24.00',
'dd-MON-YY hh24.mi.ss');

Flashback complete.
```

   As you can see, you can flash the table back and forth until you arrive at the exact point.

The flashback is complete. Since the table was not dropped, all dependent objects, such as triggers and indexes, remain unaffected.

How It Works

Table flashback is entirely different from the database flashback you saw earlier in the chapter. When a table's data changes, the past information is stored in undo segments. Oracle uses this information to present a read-consistent view of the data later. Even if the changes were committed, the undo data is important for the read-consistent image needed by a query that started after the data was changed but before it was committed.

The flashback versions query in Recipe 13-20 uses the information in the undo segments to display past versions of the data at multiple points in time. Flashing back a table uses the same undo data to reconstruct the data at whatever point in the past you specify. If sufficient information is not available in the undo segments, you will get the following error:

```
SQL> flashback table accounts to timestamp to_date ('12-JUL-12 15.23.00', 'dd-MON-YY hh24.mi.ss');
flashback table accounts to timestamp to_date ('12-JUL-12 15.23.00', 'dd-MON-YY hh24.mi.ss')
                *
ERROR at line 1:
ORA-00604: error occurred at recursive SQL level 1
ORA-12801: error signaled in parallel query server P003
ORA-01555: snapshot too old: rollback segment number 4 with name "_SYSSMU4$" too small
```

This error may not be that intuitive to interpret, but it conveys the message—the undo segment does not have information the flashback operation needs. In that case you can resort to Recipe 13-22 for an alternative mechanism to get the table back from the past.

Contrast this operation with the flashback database operation. In flashback database, the changes at the block level to the entire database are captured in flashback logs, and the flashback operation undoes the block changes. Any database change—the creation of new objects, truncation, and so on—is captured by the logs and can be played back. In a flashback query, the data is reconstructed from the undo segments. Any DDL operations are not reinstated. So if you have added a column at 1:30 p.m. and flash back to 1:25 p.m., the added column is not dropped. By the way, the DDL operation does not restrict your ability to perform a flashback beyond that point.

During the flashback operation, the database might have to move the rows from one block to another. This is allowed only if the table has the property `row movement` enabled. Therefore, you had to enable that as the first step of the process.

You can flash back a table owned by another user, but to do so you need `SELECT`, `INSERT`, `DELETE`, and `ALTER` privileges on the table, as well as one of the following:

- `FLASHBACK ANY TABLE` system privilege
- `FLASHBACK` privilege on that particular table

Table flashback does not work on the following types of tables:

- Advanced queuing (AQ) tables
- Individual table partitions or subpartitions
- Materialized views
- Nested tables
- Object tables
- Remote tables
- Static data dictionary tables
- System tables
- Tables that are part of a cluster

Some restrictions are relaxed with newer versions of the Oracle Database, so it is possible that some items may not be in this list when you read this book.

There are some important points you should know when you flash back a table. To make our description of those points easier to understand, suppose the following is a time line of events:

```
Time -> ---------------------------------------------------------------
SCN ->        1,000             2,000                   3,000         4,000
Events ->       DDL Occurred     Index Dropped     Table Data
```

The SCNs corresponding to each event are shown on the scale. The current SCN is 4,000. Given this scenario, the following limitations and caveats are true:

- You can't flash back the table to an SCN prior to SCN 1,000 (when a specific type of DDL occurred). These DDL operations are as follows:
  - Adding a constraint to the table.
  - Adding the table to a cluster.
  - Adding, dropping, merging, splitting, coalescing, or truncating a partition or subpartition. Adding a range partition is acceptable.
  - Dropping columns.
  - Modifying columns.
  - Moving the table to a different (or even the same) tablespace.
  - Truncating the table.
- When you flash back the table to a SCN prior to 2,000 (when the index was dropped), the index is not reinstated. Remember, the flashback operation is a data movement operation, not DDL, so dropped objects are not created.
- When you flash back a table, the statistics on the table are not reinstated. When you flash back the table to SCN 1,500, the statistics on the table are as of SCN 4,000.

13-22. Recovering a Specific Table from Backup

Problem

You want to restore a specific table, not the entire database. The table is no longer in the recycle bin, but you do have the RMAN backup.

Solution

Oracle Database 12.1 introduces a new feature that enables you to recover a table or a partition from an RMAN backup. You can restore just the table or partion, without needing to restore the entire, containing tablespace.

Here are the steps to recover a table named ACCOUNTS in the SCOTT schema:

1. Make sure the database is in archivelog mode:

```
SQL> select log_mode from v$database;

LOG_MODE
------------
ARCHIVELOG
```

2. Make sure the database is open in read/write mode:

```
SQL> select open_mode from v$database;

OPEN_MODE
----------
READ WRITE
```

3. Choose where a temporary database can be created for the duration of the RMAN operation. Designate a file system or an ASM disk group–the choice is yours. Just be sure you choose a destination having enough space to hold SYSTEM, SYSAUX, and Undo tablespaces along with the tablespace that holds the table. In this example, we will use the disk group DG1.

4. Decide the point in time to restore the table to. You can specify the point in time using either of:

   a. The timestamp

   b. The SCN

   In this example, in step 4, we specify recovery up to one minute prior to the current time. Since a day has 24 hours, and an hour has 60 minutes, the expression `sysdate - 1/60/24` represents the time just one minute ago.

   Pay careful attention to choosing the time. The table you are recovering must have been present in the database at that time. If the table was dropped prior to that time, the recovery will fail with "`RMAN-05057: Table not found`" error.

5. Connect to RMAN:

```
$ rman
RMAN> connect target '/ as sysdba'
```

■ **Caution** As of the writing of this book, there is bug #14172827 that causes the last step of this process to fail. To avoid that bug, connect to the target database in RMAN as sysdba or as SYS, i.e., `connect target '/ as sysdba'`, or as `connect target sys/<SysPassword>; do` not connect as `connect target /`.

---

6. Issue the following command from RMAN. Be sure to plug in your chosen point in time:

```
RMAN> recover table SCOTT.ACCOUNTS
2> until time 'sysdate-1/60/24'
3> auxiliary destination '+DG1'
4> ;
```

   The command will execute with a long output, which is not reproduced here for the sake of brevity. At the end, the table will be reinstated in the database from the backup.

   If you know the SCN, you can use that instead of the timestamp. For example:

```
RMAN> recover table SCOTT.ACCOUNTS
2> until scn 2012991
3> auxiliary destination '+DG1'
4> ;
```

■ **Note** If you have one or more pluggable databases on a container database, the `recover table` command works only when connected to the container database, not to the pluggable ones.

How It Works

From the earlier recipes you learned how to recover a table manually from backup. In summary here are the high-level steps in that operation:

1. Create another database instance.
2. Restore from the backup the tablespace containing that table along with system, sysaux and undo tablespaces in a different location.
3. Open that restored database.
4. Export the table.
5. Import into the main database.
6. Drop this temporary database and delete the instance.

The recover table operation does all this work behind the scenes without you having to worry about the commands and other details. If you examine the very long output after the command, you can see the precise commands used by RMAN.

Let's examine some of the output from the `recover` command:

1. A temporary instance is created:

```
Creating automatic instance, with SID='ykFp'
 initialization parameters used for automatic instance:
db_name=CDB1
db_unique_name=ykFp_pitr_CDB1
```

   RMAN chooses a random string as SID so as not to clash with an existing SID. The unique name of the database ykFp_pitr_CDB1 is also another way to avoid using one of the existing database names.

2. A clone control file is mounted:

```
contents of Memory Script:
{
# set requested point in time
set until  time "sysdate-1/60/24";
# restore the controlfile
restore clone controlfile;
# mount the controlfile
sql clone 'alter database mount clone database';
# archive current online log
sql 'alter system archive log current';
}
```

3. Relevant data files are restored:

```
{
# set requested point in time
set until  time "sysdate-1/60/24";
# set destinations for recovery set and auxiliary set datafiles
set newname for clone datafile  1 to new;
set newname for clone datafile  4 to new;
set newname for clone datafile  12 to new;
set newname for clone datafile  3 to new;
set newname for clone tempfile  1 to new;
# switch all tempfiles
switch clone tempfile all;
# restore the tablespaces in the recovery set and the auxiliary set
restore clone datafile  1, 4, 12, 3;
switch clone datafile all;
}
```

4. The relevant data files are restored to the location specified by the auxiliary parameter in the RMAN recover table command—DG1:

```
channel ORA_AUX_DISK_1: starting datafile backup set restore
channel ORA_AUX_DISK_1: specifying datafile(s) to restore from backup set
channel ORA_AUX_DISK_1: restoring datafile 00001 to +DG1
channel ORA_AUX_DISK_1: restoring datafile 00004 to +DG1
channel ORA_AUX_DISK_1: restoring datafile 00012 to +DG1
channel ORA_AUX_DISK_1: restoring datafile 00003 to +DG1
channel ORA_AUX_DISK_1: reading from backup piece
+FRA/cdb1/backupset/2012_08_04/nnndf0_tag20120804t133222_0.427.790435943
channel ORA_AUX_DISK_1: piece
handle=+FRA/cdb1/backupset/2012_08_04/nnndf0_tag20120804t133222_0.427.790435943
tag=TAG20120804T133222
channel ORA_AUX_DISK_1: restored backup piece 1
channel ORA_AUX_DISK_1: restore complete, elapsed time: 00:00:35
Finished restore at 04-AUG-12
```

5. The clone database is recovered:

```
contents of Memory Script:
{
# set requested point in time
set until  time "sysdate-1/60/24";
# online the datafiles restored or switched
sql clone "alter database datafile  1 online";
sql clone "alter database datafile  4 online";
sql clone "alter database datafile  12 online";
sql clone "alter database datafile  3 online";
# recover and open database read only
recover clone database tablespace  "SYSTEM", "UNDOTBS1", "UNDOTBS2", "SYSAUX";
sql clone 'alter database open read only';
}
```

6. Media recovery starts and completes on these data files:

```
starting media recovery

archived log for thread 1 with sequence 67 is already on disk as file
+FRA/cdb1/archivelog/2012_08_04/thread_1_seq_67.420.790436095
archived log for thread 1 with sequence 68 is already on disk as file
+FRA/cdb1/archivelog/2012_08_04/thread_1_seq_68.425.790436113
archived log for thread 1 with sequence 69 is already on disk as file
+FRA/cdb1/archivelog/2012_08_04/thread_1_seq_69.423.790448453
archived log file name=+FRA/cdb1/archivelog/2012_08_04/thread_1_seq_67.420.790436095
thread=1 sequence=67
archived log file name=+FRA/cdb1/archivelog/2012_08_04/thread_1_seq_68.425.790436113
thread=1 sequence=68
archived log file name=+FRA/cdb1/archivelog/2012_08_04/thread_1_seq_69.423.790448453
thread=1 sequence=69
media recovery complete, elapsed time: 00:00:02
Finished recover at 04-AUG-12

sql statement: alter database open read only
```

7. The table is exported via Data Pump:

```
contents of Memory Script:
{
# create directory for datapump import
sql "create or replace directory TSPITR_DIROBJ_DPDIR as ''
+DG1''";
# create directory for datapump export
sql clone "create or replace directory TSPITR_DIROBJ_DPDIR as ''
+DG1''";
}
executing Memory Script

sql statement: create or replace directory TSPITR_DIROBJ_DPDIR as ''+DG1''

sql statement: create or replace directory TSPITR_DIROBJ_DPDIR as ''+DG1''

Performing export of tables...
   EXPDP> Starting "SYS"."TSPITR_EXP_dscb_CvAe":
   EXPDP> Estimate in progress using BLOCKS method...
   EXPDP> Processing object type TABLE_EXPORT/TABLE/TABLE_DATA
```

8. The table is imported into the main database:

```
Performing import of tables...
   IMPDP> Master table "SYS"."TSPITR_IMP_dscb_zhwf" successfully loaded/unloaded
   IMPDP> Starting "SYS"."TSPITR_IMP_dscb_zhwf":
   IMPDP> Processing object type TABLE_EXPORT/TABLE/TABLE
   IMPDP> Processing object type TABLE_EXPORT/TABLE/TABLE_DATA
   IMPDP> . . imported "SCOTT"."ACCOUNTS"                          5.031 KB
```

9. Finally, the temporary instance is dropped:

```
Removing automatic instance
shutting down automatic instance
Oracle instance shut down
Automatic instance removed
auxiliary instance file +DG1/cdb1/datafile/sysaux.259.790448475 deleted
auxiliary instance file +DG1/cdb1/datafile/undotbs2.257.790448475 deleted
... and so on ...
```

If you encounter an error at any of the steps, it will be clearly visible in the above steps. The most common issues that may come up during this activity are:

- Backup does not have the tablespace that contains the table to be recovered
- The table was not present at the SCN or timestamp given in the recover command
- Not enough storage to restore the auxiliary database
- Not enough memory to create the auxiliary instance

Here are the restrictions on the recover table process:

- SYS owned tables can't be recovered.
- The `recover table` works by performing a point in time recovery of the tablespace, which is not allowed for SYSTEM and SYSAUX. Therefore, tables in these two tablespaces can't be recovered with this command.
- Tables can't be recovered on a physical standby database.

13-23. Recovering a Partition

Problem

You want to recover a single partition named P1 from a table named ACCOUNTS from the backup.

Solution

Follow the prerequisites explained in Recipe 13-22—e.g., there should be enough space in the auxiliary destination to hold the tablespaces SYSTEM, SYSAUX, and the tablespace where the partition exists. Sufficient memory to run another instance and the backup that contains the partition. After ensuring all those prerequisites are satisfied, follow these steps in RMAN:

1. Connect to RMAN target database:

```
RMAN> connect target "/ as sysdba"
```

2. Recover the table's partition with this command:

```
RMAN> recover table  scott.accounts:P1
2> until scn 1799975
3> auxiliary destination '+DG1';
```

This creates a table called ACCOUNTS_P1, which is a replica of the partition P1 of the table ACCOUNTS.

3. Create an empty partition on the table. In this case, assume there is a partition called P2. So, you will need to split the partition P2.

```
SQL> alter table accounts
  2  split partition p2
  3  at (102)
  4  into
  5  (
  6     partition p1,
  7     partition p2
  8* );

Table altered.
```

   This splits the partition P2 into two partitions: P1 and P2. All the data in P2 are still there and P1 is completely empty.

4. Swap the newly recovered table with this empty partition:

```
SQL> alter table accounts
  2  exchange partition p1
  3  with table accounts_p1
  4  without validation;

Table altered.
```

   Now the partition P1 contains the data from the backup.

5. The table ACCOUNTS_P1 is now empty. Drop the table:

```
SQL> drop table accounts_p1;

Table dropped.
```

How It Works

The recovery of a partition follows the same mechanism as the recovery of a table from the backup of the database as shown in Recipe 13-22. The only difference is that the recovered partitions are created not as partitions, but rather as independent tables. The syntax of the command used in step 2 of the solution is:

```
recover table <Owner>.<TableName>:<PartitionName>
```

If more than one partition is to be rcovered, you can mention them separated by commas:

```
recover table scott.accounts:P1, scott.accounts:P2
```

Each partition is recovered as a separate table with the naming convention as <TableName>_<PartitionName>. The partition P1 of the table ACCOUNTS will be recovered as ACCOUNTS_P1. If you recover multiple partitions of a table, each partition is imported into an individual table.

13-24. Recovering a Table into a Different Name

Problem

You want to recover a table, but a table with the same name already exists in the database.

Solution

Suppose you want to recover a table called ACCOUNTS but there is already a table with that same name. Thus, you want to recover the table under a new name: ACCOUNTS_NEW. Ensure the prerequisites explained in Recipe 13-22. As explained in that recipe, use the `recover table` command, but with a little addition: the `remap table` clause.

1. Connect to RMAN:

```
$ rman
RMAN> connect target "/ as sysdba"
```

2. Recover the table but with `remap table` clause that creates a table with a different name—ACCOUNTS_NEW:

```
RMAN> recover table  arup.accounts
2> until scn 1799975
3> auxiliary destination '+DG1'
4> remap table arup.accounts:accounts_new;
```

The remap clause at the end causes the table ACCOUNTS to be restored with the name ACCOUNTS_NEW.

How It Works

Recall from Recipe 13-22 that the recover command creates a temporary instance and recovers just the tablespaces that are required for the table. The operation then exports the table and imports it into the main database. The `remap table` clause injects the REMAP_TABLE option in the final import process to import the table into a new name. The `remap table` clause in the `recover` command does the trick of restoring the table in the new name.

Here is the syntax of the remap table clause:

```
remap table <Owner>.<OldTableName>:<NewTableName>
```

If you want to recover a partition of the table:

```
remap table <Owner>.<OldTableName>:<PartitionName>:<NewTableName>
```

So, had you wanted to recover a partition P1 of table ACCOUNTS to a new table called ACCOUNTS_P1_NEW, you would have used the following clause instead:

```
remap table arup.accounts:p1:accounts_p1_new;
```

When you recover a partition, it always goes to a separate table.

13-25. Recovering a Table into a Different Tablespace

Problem

You want to recover a table, but not to the same tablespace it was in earlier. You don't want the default tablespace, but rather a different tablespace you choose during the recovery operation.

Solution

Follow the prerequisites explained in Recipe 13-22 and recover the table, but with a small change. The `recover table` statement has a `remap tablespace` clause that allows substituting tablespace names. Suppose you want to recover a table ACCOUNTS but want it to go to ACCDATA tablespace; not USERS. To do that, follow these steps:

1. Connect to RMAN:

```
$ rman
RMAN> connect target "/ as sysdba"
```

2. Recover the table partition but with remap clause:

```
RMAN> recover table  arup.accounts
2> until scn 1799975
3> auxiliary destination '+DG1'
4> remap tablespace users:accdata;
```

The last clause imports the table ACCOUNTS into the database, but instead of the default tablespace, it imports the table into the tablespace ACCDATA.

How It Works

The recipe works the same way as the Recipe 13-22. The `remap tablespace` clause in `recover` command does the trick. Here is the syntax of the `remap` clause:

```
remap tablespace <OldTablespace>:<NewTablespace>
```

The usual preconditions apply to the recover option here, e.g., the tablespace should have sufficient free space, etc. Note from the "How It Works" section of Recipe 13-22 that the recover command creates a new auxiliary instance, recovers just enough tablespaces necessary to recover the table, and exports that table and imports into the main database. The remap tablespace clause alters the IMPDP command to include the `remap_tablespace` option, which causes the process to import the table to a different tablespace.

13-26. Creating an Export Dump of a Table to be Recovered

Problem

You want to recover a table from backup, but not in the form of a table in a database. Instead, you want to create a Data Pump Export dump file with the table structure and data. What you want is the ability *later* to import the table and data into any database you choose.

Solution

Follow the prerequisites mentioned in Recipe 13-22. Then do the following:

1. Identify the location where the dump file will be created. In this example, we chose '/tmp' files system. You do not need to create a database directory object on this Unix directory.
2. Choose the name of the dump file. In this case we chose the name accounts.dmp.
3. Connect to RMAN:

```
$ rman
RMAN> connect target "/ as sysdba"
```

4. Issue the following command from RMAN prompt:

```
RMAN> recover table scott.accounts
2> until scn 1792736
3> auxiliary destination '+DG1'
4> datapump destination '/tmp' dump file 'accounts.dmp'
5> notableimport;
```

5. Check the existence of the dump file at the location specified - /tmp:

```
$ ls -l /tmp/accounts.dmp
```

Now you can use this file to import into a different database, or the same database later.

How It Works

The mechanics of the process has been described in Recipe 13-22. Like that recipe, this recipe's solution creates a temporary instance, recovers the tablespaces necessary for the table ACCOUNTS, and exports the table. But unlike that other recipe, the solution here does not import the table from that dump file.

Note the additional clause `notableimport`, which causes the table not to be imported into the database. Instead RMAN leaves the table in the dump file called accounts.dmp in the directory /tmp.

When you use the `notableimport` clause, you can't use `remap tablespace` or `remap table`. When you have the dump file, you can use it to import the table into any tablespace. If needed, you can rename the table as part of issuing the `impdp` command, so the remap clauses are not relevant and hence not available.