**CHAPTER 8  
Changing Table Contents**

In this chapter, you’ll learn more about changing the contents of tables. Specifically, you’ll learn the following:

Image How to add, modify, and remove rows using the INSERT, UPDATE, and DELETE statements

Image How database transactions may consist of multiple INSERT, UPDATE, and DELETE statements

Image How to make the results of your transactions permanent using the COMMIT statement or undo their results entirely using the ROLLBACK statement

Image How an Oracle database can process multiple transactions at the same time

Image How to use query flashbacks to view rows as they originally were before you made changes to them

**ADDING ROWS USING THE INSERT STATEMENT**

You use the INSERT statement to add rows to a table. You can specify the following information in an INSERTstatement:

Image The table into which the row is to be inserted

Image A list of columns for which you want to specify column values

Image A list of values to store in the specified columns

When adding a row, you typically supply a value for the primary key and all other columns that are defined as NOT NULL. You don’t have to specify values for NULL columns if you don’t want to; by default they will be set to null.

You can find out which columns are defined as NOT NULL using the SQL\*Plus DESCRIBE command. The following example describes the customers table:

**DESCRIBE customers**



As you can see, the customer\_id, first\_name, and last\_name columns are NOT NULL, meaning that you must supply a value for these columns. The dob and phone columns don’t require a value: If you omit these values when adding a row, the columns would be set to null.

The following INSERT statement adds a row to the customers table. Notice that the order of values in the VALUES clause matches the order in which the columns are specified in the column list. Also notice that the statement has three parts: the table name, the column list, and the values to be added:

**INSERT INTO customers (**

**customer\_id, first\_name, last\_name, dob, phone**

**) VALUES (**

**6, 'Fred', 'Brown', '01-JAN-1970', '800-555-1215'**

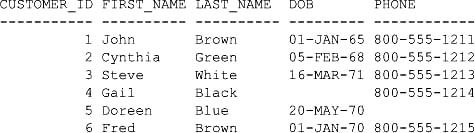
**);**

1 row created.

SQL\*Plus responds that one row has been created. You can verify this by performing the following SELECTstatement:

**SELECT \***

**FROM customers;**



Notice the new row appears in the results returned by the query.

**Omitting the Column List**

You may omit the column list when supplying values for every column, as in this example:

**INSERT INTO customers**

**VALUES (7, 'Jane', 'Green', '01-JAN-1970', '800-555-1216');**

When you omit the column list, the order of the values you supply must match the order of the columns as listed in the output from the DESCRIBE command.

**Specifying a Null Value for a Column**

You can specify a null value for a column using the NULL keyword. For example, the following INSERTspecifies a null value for the dob and phone columns:

**INSERT INTO customers**

**VALUES (8, ’Sophie', 'White', NULL, NULL);**

When you view this row using a query, you won’t see a value for the dob and phone columns, because they’ve been set to null values:

**SELECT \***

**FROM customers**

**WHERE customer\_id = 8;**

Image

Notice the dob and phone column values are blank.

**Including Single and Double Quotes in a Column Value**

You can include a single and double quote in a column value. For example, the following INSERT specifies a last name of O'Malley for a new customer; notice the use of two single quotes in the last name after the letter O:

**INSERT INTO customers**

**VALUES (9, 'Kyle', 'O''Malley', NULL, NULL);**

The next example specifies the name The "Great" Gatsby for a new product:

**INSERT INTO products (**

**product\_id, product\_type\_id, name, description, price**

**) VALUES (**

**13, 1, 'The "Great" Gatsby', NULL, 12.99**

**);**

**Copying Rows from One Table to Another**

You can copy rows from one table to another using a query in the place of the column values in the INSERTstatement. The number of columns and the column types in the source and destination must match. The following example uses a SELECT to retrieve the first\_name and last\_name columns for customer #1 and supplies those columns to an INSERT statement:

**INSERT INTO customers (customer\_id, first\_name, last\_name)**

**SELECT 10, first\_name, last\_name**

**FROM customers**

**WHERE customer\_id = 1;**

Notice that the customer\_id for the new row is set to 10.

Image

**NOTE**  
*Oracle Database 9* i *introduced the* MERGE *statement, which allows you to merge rows from one table to another*. MERGE *is much more flexible than combining an* INSERT *and a* SELECT *to copy rows from one table to another. You’ll learn about* MERGE *later in the section "Merging Rows Using MERGE."*

**MODIFYING ROWS USING THE UPDATE STATEMENT**

You use the UPDATE statement to modify rows in a table. When you use the UPDATE statement, you typically specify the following information:

Image The table name

Image A WHERE clause that specifies the rows to be changed

Image A list of column names, along with their new values, specified using the SET clause

You can change one or more rows using the same UPDATE statement. If more than one row is specified, the same change will be implemented for all of those rows. For example, the following UPDATE statement sets the last\_name column to Orange for the row whose customer\_id is 2:

**UPDATE customers**

**SET last\_name = 'Orange'**

**WHERE customer\_id = 2;**

1 row updated.

SQL\*Plus confirms that one row was updated. If the WHERE clause were omitted, then all the rows would be updated. The following query confirms the change was made:

**SELECT \***

**FROM customers**

**WHERE customer\_id = 2;**

Image

You can change multiple rows and multiple columns in the same UPDATE statement. For example, the following UPDATE raises the price by 20 percent for all products whose current price is greater than or equal to $20. The UPDATE also changes those products' names to lowercase:

**UPDATE products**

**SET**

**price = price \* 1.20**,

**name = LOWER(name)**

**WHERE**

**price >= 20;**

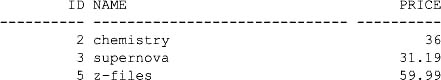
3 rows updated.

As you can see, three rows are updated by this statement. The following query confirms the change:

**SELECT product\_id, name, price**

**FROM products**

**WHERE price >= (20 \* 1.20);**



Image

**NOTE**  
*You can also use a subquery with an* UPDATE *statement. This is covered in*[*Chapter 6*](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch06.html#ch06)*in the section "Writing an UPDATE Statement Containing a Subquery."*

**THE RETURNING CLAUSE**

In Oracle Database 10*g* and above, you can use the RETURNING clause to return the value from an aggregate function such as AVG(). Aggregate functions were covered in [Chapter 4](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#ch04).

The following tasks are performed by the next example:

Image Declares a variable named average\_product\_price

Image Decreases the price column of the rows in the products table and saves the average price in the average\_product\_price variable using the RETURNING clause

Image Prints the value of the average\_product\_price variable

**VARIABLE average\_product\_price NUMBER**

**UPDATE products**

**SET price = price \* 0.75**

**RETURNING AVG(price) INTO :average\_product\_price;**

12 rows updated.

**PRINT average\_product\_price**

AVERAGE\_PRODUCT\_PRICE

------------------------

16.1216667

**REMOVING ROWS USING THE DELETE STATEMENT**

You use the DELETE statement to remove rows from a table. Generally, you should specify a WHERE clause that limits the rows that you wish to delete; if you don’t, *all* the rows will be deleted.

The following DELETE statement removes the row from the customers table whose customer\_id is 10:

**DELETE FROM customers**

**WHERE customer\_id = 10;**

1 row deleted.

SQL\*Plus confirms that one row has been deleted.

You can also use a subquery with a DELETE statement. This is covered in [Chapter 6](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch06.html#ch06) in the section "Writing a DELETE Statement Containing a Subquery."

Image

**NOTE**  
*If you’ve been following along with the previous* INSERT, UPDATE, *and* DELETE *statements, roll them back using* ROLLBACK *so that your results match those shown in the rest of this chapter. Don’t worry if you’ve already disconnected from the database: simply rerun the* store\_schema.sql*script to re-create everything*.

**DATABASE INTEGRITY**

When you execute a DML statement (an INSERT, UPDATE, or DELETE, for example), the database ensures that the rows in the tables maintain their integrity. This means that any changes made to the rows do not affect the primary key and foreign key relationships for the tables.

**Enforcement of Primary Key Constraints**

Let’s examine some examples that show the enforcement of a primary key constraint. The customers table’s primary key is the customer\_id column, which means that every value stored in the customer\_id column must be unique. If you try to insert a row with a duplicate value for a primary key, the database returns the error ORA-00001, as in this example:

SQL> **INSERT INTO customers (**

2 **customer\_id, first\_name, last\_name, dob, phone**

3 **) VALUES (**

4 **1, 'Jason', 'Price', '01-JAN-60', '800-555-1211'**

5 **);**

INSERT INTO customers (

\*

ERROR at line 1:

ORA-00001: unique constraint (STORE.CUSTOMERS\_PK) violated

If you attempt to update a primary key value to a value that already exists in the table, the database returns the same error:

SQL> **UPDATE customers**

2 **SET customer\_id = 1**

3 **WHERE customer\_id = 2;**

UPDATE customers

\*

ERROR at line 1:

ORA-00001: unique constraint (STORE.CUSTOMERS\_PK) violated

**Enforcement of Foreign Key Constraints**

A foreign key relationship is one in which a column from one table is referenced in another. For example, the product\_type\_id column in the products table references the product\_type\_id column in the product\_types table. The product\_types table is known as the *parent* table, and the products table is known as the *child* table, reflecting the dependence of the product\_type\_id column in the products table on the product\_type\_id column in the product\_types table.

If you try to insert a row into the products table with a nonexistent product\_type\_id, the database will return the error ORA-02291. This error indicates the database couldn’t find a matching parent key value (the parent key is the product\_type\_id column of the product\_types table). In the following example, the error is returned because there is no row in the product\_types table whose product\_type\_id is 6:

SQL> **INSERT INTO products (**

2 **product\_id, product\_type\_id, name, description, price**

3 **) VALUES (**

4 **13, 6, 'Test', 'Test', NULL**

5 **);**

INSERT INTO products (

\*

ERROR at line 1:

ORA-02291: integrity constraint (STORE.PRODUCTS\_FK\_PRODUCT\_TYPES)

violated - parent key not found

Similarly, if you attempt to update the product\_type\_id of a row in the products table to a nonexistent parent key value, the database returns the same error, as in this example:

SQL> **UPDATE products**

2 **SET product\_type\_id = 6**

3 **WHERE product\_id = 1;**

UPDATE products

\*

ERROR at line 1:

ORA-02291: integrity constraint (STORE.PRODUCTS\_FK\_PRODUCT\_TYPES)

violated - parent key not found

Finally, if you attempt to delete a row in the parent table that has dependent child rows, the database returns error ORA-02292. For example, if you attempt to delete the row whose product\_type\_id is 1 from the product\_types table, the database will return this error because the products table contains rows whose product\_type\_id is 1:

SQL> **DELETE FROM product\_types**

2 **WHERE product\_type\_id = 1;**

DELETE FROM product\_types

\*

ERROR at line 1:

ORA-02292: integrity constraint (STORE.PRODUCTS\_FK\_PRODUCT\_TYPES)

violated - child record found

If the database were to allow this deletion, the child rows would be invalid because they wouldn’t point to valid values in the parent table.

**USING DEFAULT VALUES**

Oracle Database 9*i* introduced a feature that allows you to define a default value for a column. For example, the following statement creates a table named order\_status; the status column is defaulted to 'Order placed' and the last\_modified column is defaulted to the date and time returned by SYSDATE:

CREATE TABLE order\_status (

order\_status\_id INTEGER

CONSTRAINT default\_example\_pk PRIMARY KEY,

status VARCHAR2(20) DEFAULT 'Order placed' NOT NULL,

last\_modified DATE DEFAULT SYSDATE

);

Image

**NOTE**  
*The* order\_status *table is created by the* store\_schema.sql *script. This means you don’t have to type in the previous* CREATE TABLE *statement yourself. Also, you don’t have to type in the*INSERT *statements shown in this section*.

When you add a new row to the order\_status table but don’t specify the values for the status and last\_modified columns, those columns are set to the default values. For example, the following INSERTstatement omits values for the status and last\_modified columns:

INSERT INTO order\_status (order\_status\_id)

VALUES (1);

The status column is set to the default value of 'Order placed', and the last\_modified column is set to the current date and time.

You can override the defaults by specifying a value for the columns, as shown in the following example:

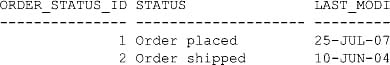
INSERT INTO order\_status (order\_status\_id, status, last\_modified)

VALUES (2, 'Order shipped', '10-JUN-2004');

The following query retrieves the rows from order\_status:

**SELECT \***

**FROM order\_status;**



You can set a column back to the default using the DEFAULT keyword in an UPDATE statement. For example, the following UPDATE sets the status column to the default:

**UPDATE order\_status**

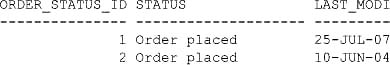
**SET status = DEFAULT**

**WHERE order\_status\_id = 2;**

The following query shows the change made by this UPDATE statement:

**SELECT \***

**FROM order\_status;**



**MERGING ROWS USING MERGE**

Oracle Database 9*i* introduced the MERGE statement, which allows you to merge rows from one table into another. For example, you might want to merge changes to products listed in one table into the products table.

The store schema contains a table named product\_changes that was created using the following CREATE TABLE statement in store\_schema.sql:

CREATE TABLE product\_changes (

product\_id INTEGER

CONSTRAINT prod\_changes\_pk PRIMARY KEY,

product\_type\_id INTEGER

CONSTRAINT prod\_changes\_fk\_product\_types

REFERENCES product\_types(product\_type\_id),

name VARCHAR2(30) NOT NULL,

description VARCHAR2(50),

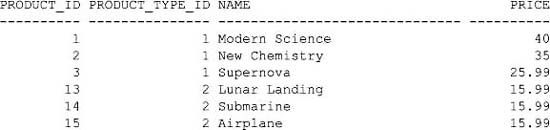
price NUMBER(5, 2)

);

The following query retrieves the product\_id, product\_type\_id, name, and price columns from this table:

**SELECT product\_id, product\_type\_id, name, price**

**FROM product\_changes;**



Let’s say you want to merge the rows from the product\_changes table into the products table as follows:

Image For rows with matching product\_id values in the two tables, update the existing rows in products with the column values from product\_changes. For example, product #1 has a different price in product\_changes from the one in products; therefore, product #1’s price must be updated in the products table. Similarly, product #2 has a different name and price, so both values must be updated in products. Finally, product #3 has a different product\_type\_id, and so this value must be updated in products.

Image For new rows in product\_changes, insert those new rows into the products table. Products #13, #14, and #15 are new in product\_changes and must therefore be inserted into products.

The easiest way to learn how to use the MERGE statement is to see an example. The following example performs the merge as defined in the previous bullet points:

**MERGE INTO products p**

**USING product\_changes pc ON (**

**p.product\_id = pc.product\_id**

**)**

**WHEN MATCHED THEN**

**UPDATE**

**SET**

**p.product\_type\_id = pc.product\_type\_id**,

**p.name = pc.name**,

**p.description = pc.description**,

**p.price = pc.price**

**WHEN NOT MATCHED THEN**

**INSERT (**

**p.product\_id, p.product\_type\_id, p.name**,

**p.description, p.price**

**) VALUES (**

**pc.product\_id, pc.product\_type\_id, pc.name**,

**pc.description, pc.price**

**);**

6 rows merged.

Image

**NOTE**  
*You’ll find a script named* merge\_example.sql *in the* SQL *directory. This script contains the previous* MERGE *statement*.

Notice the following points about the MERGE statement:

Image The MERGE INTO clause specifies the name of the table to merge the rows into. In the example, this is the products table, which is given an alias of p.

Image The USING … ON clause specifies a table join. In the example, the join is made on the product\_idcolumns in the products and product\_changes tables. The product\_changes table is also given an alias of pc.

Image The WHEN MATCHED THEN clause specifies the action to take when the USING … ON clause is satisfied for a row. In the example, this action is an UPDATE statement that sets the product\_type\_id, name, description, and price columns of the existing row in the products table to the column values for the matching row in the product\_changes table.

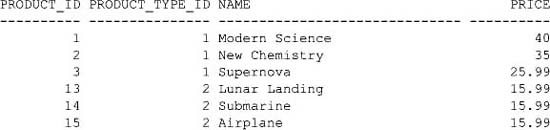
Image The WHEN NOT MATCHED THEN clause specifies the action to take when the USING … ON clause is *not*satisfied for a row. In the example, this action is an INSERT statement that adds a row to the products table, taking the column values from the row in the product\_changes table.

If you run the previous MERGE statement, you’ll see that it reports six rows are merged; these are the rows with product\_id values of 1, 2, 3, 13, 14, and 15. The following query retrieves the six merged rows from the products table:

**SELECT product\_id, product\_type\_id, name, price**

**FROM products**

**WHERE product\_id IN (1, 2, 3, 13, 14, 15);**



The following changes were made to these rows:

Image Product #1 has a new price.

Image Product #2 has a new name and price.

Image Product #3 has a new product type ID.

Image Products #13, #14, and #15 are new.

Now that you’ve seen how to make changes to the contents of tables, let’s move on to database transactions.

**DATABASE TRANSACTIONS**

A database *transaction* is a group of SQL statements that perform a *logical unit of work*. You can think of a transaction as an inseparable set of SQL statements whose results should be made permanent in the database as a whole (or undone as a whole).

An example of a database transaction is a transfer of money from one bank account to another. One UPDATEstatement would subtract from the total amount of money from one account, and another UPDATE would add money to the other account. Both the subtraction and the addition must be permanently recorded in the database; otherwise, money will be lost. If there is a problem with the money transfer, then the subtraction and addition must both be undone. The simple example outlined in this paragraph uses only two UPDATE statements, but a transaction may consist of many INSERT, UPDATE, and DELETE statements.

**Committing and Rolling Back a Transaction**

To permanently record the results made by SQL statements in a transaction, you perform a *commit*, using the SQL COMMIT statement. If you need to undo the results, you perform a *rollback*, using the SQL ROLLBACKstatement, which resets all the rows back to what they were originally.

The following example adds a row to the customers table and then makes the change permanent by performing a COMMIT:

**INSERT INTO customers**

**VALUES (6, 'Fred', 'Green', '01-JAN-1970', '800-555-1215');**

1 row created.

**COMMIT;**

Commit complete.

The following example updates customer #1 and then undoes the change by performing a ROLLBACK:

**UPDATE customers**

**SET first\_name = 'Edward'**

**WHERE customer\_id = 1;**

1 row updated.

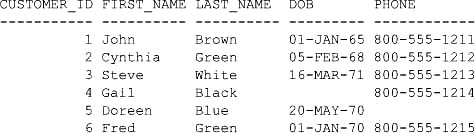
**ROLLBACK;**

Rollback complete.

The following query shows the new row from the COMMIT statement:

**SELECT \***

**FROM customers;**



Notice that customer #6 has been made permanent by the COMMIT, but the change to customer #1’s first name has been undone by the ROLLBACK.

**Starting and Ending a Transaction**

A transaction is a logical unit of work that enables you to split up your SQL statements. A transaction has a beginning and an end; it begins when one of the following events occurs:

Image You connect to the database and perform a DML statement (an INSERT, UPDATE, or DELETE).

Image A previous transaction ends and you enter another DML statement.

A transaction ends when one of the following events occurs:

Image You perform a COMMIT or a ROLLBACK.

Image You perform a DDL statement, such as a CREATE TABLE statement, in which case a COMMIT is automatically performed.

Image You perform a DCL statement, such as a GRANT statement, in which case a COMMIT is automatically performed. You’ll learn about GRANT in the next chapter.

Image You disconnect from the database. If you exit SQL\*Plus normally, by entering the EXIT command, a COMMITis automatically performed for you. If SQL\*Plus terminates abnormally—for example, if the computer on which SQL\*Plus was running were to crash—a ROLLBACK is automatically performed. This applies to any program that accesses a database. For example, if you wrote a Java program that accessed a database and your program crashed, a ROLLBACK would be automatically performed.

Image You perform a DML statement that fails, in which case a ROLLBACK is automatically performed for that individual DML statement.

Image

**TIP**  
*It is poor practice not to explicitly commit or roll back your transactions, so perform a* COMMIT *or*ROLLBACK *at the end of your transactions*.

**Savepoints**

You can also set a *savepoint* at any point within a transaction. These allow you to roll back changes to that savepoint. Savepoints can be useful to break up very long transactions, because, if you make a mistake after you’ve set a savepoint, you don’t have to roll back the transaction all the way to the start. However, you should use savepoints sparingly: you might be better off restructuring your transaction into smaller transactions instead.

You’ll see an example of a savepoint shortly, but first let’s see the current price for products #4 and #5:

**SELECT product\_id, price**

**FROM products**

**WHERE product\_id IN (4, 5);**

Image

The price for product #4 is $13.95, and the price for product #5 is $49.99. The following UPDATE increases the price of product #4 by 20 percent:

**UPDATE products**

**SET price = price \* 1.20**

**WHERE product\_id = 4;**

1 row updated.

The following statement sets a savepoint named save1:

**SAVEPOINT save1;**

Savepoint created.

Any DML statements run after this point can be rolled back to the savepoint, and the change made to product #4 will be kept.

The following UPDATE increases the price of product #5 by 30 percent:

**UPDATE products**

**SET price = price \* 1.30**

**WHERE product\_id = 5;**

1 row updated.

The following query gets the prices of the two products:

**SELECT product\_id, price**

**FROM products**

**WHERE product\_id IN (4, 5);**

Image

Product #4’s price is 20 percent greater, and product #5’s price is 30 percent greater. The following statement rolls back the transaction to the savepoint established earlier:

**ROLLBACK TO SAVEPOINT save1;**

Rollback complete.

This has undone the price change for product #5, but left the price change for product #4 intact. The following query shows this:

**SELECT product\_id, price**

**FROM products**

**WHERE product\_id IN (4, 5);**

Image

As expected, product #4 has kept its increased price, but product #5’s price is back to the original. The following ROLLBACK undoes the entire transaction:

**ROLLBACK;**

Rollback complete.

This has undone the change made to product #4’s price, as is shown by the following query:

**SELECT product\_id, price**

**FROM products**

**WHERE product\_id IN (4, 5);**

Image

**ACID Transaction Properties**

Earlier, I defined a transaction as being a *logical unit of work*, that is, a group of related SQL statements that are either *committed* or *rolled back* as one unit. Database theory’s more rigorous definition of a transaction states that a transaction has four fundamental properties, known as *ACID* properties (from the first letter of each property in the following list):

Image **Atomic** Transactions are atomic, meaning that the SQL statements contained in a transaction make up a single unit of work.

Image **Consistent** Transactions ensure that the database state remains consistent, meaning that the database is in a consistent state when a transaction begins and that it ends in another consistent state when the transaction finishes.

Image **Isolated** Separate transactions should not interfere with each other.

Image **Durable** Once a transaction has been committed, the database changes are preserved, even if the machine on which the database software is running crashes later.

The Oracle database software handles these ACID properties and has extensive recovery facilities for restoring databases after system crashes.

**Concurrent Transactions**

The Oracle database software supports many users interacting with a database, and each user can run their own transactions at the same time. These transactions are known as *concurrent* transactions.

If users are running transactions that affect the same table, the effects of those transactions are separated from each other until a COMMIT is performed. The following sequence of events, based on two transactions named T1 and T2 that access the customers table, illustrates the separation of transactions:

**1**. T1 and T2 perform a SELECT that retrieves all the rows from the customers table.

**2**. T1 performs an INSERT to add a row in the customers table, but T1 doesn’t perform a COMMIT.

**3**. T2 performs another SELECT and retrieves the same rows as those in step 1. T2 doesn’t "see" the new row added by T1 in step 2.

**4**. T1 finally performs a COMMIT to permanently record the new row added in step 2.

**5**. T2 performs another SELECT and finally "sees" the new row added by T1.

To summarize: T2 doesn’t see the changes made by T1 until T1 commits its changes. This is the default level of isolation between transactions, but, as you’ll learn later in the section "Transaction Isolation Levels," you can change the level of isolation.

[Table 8-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch08.html#table_8-1) shows sample SQL statements that further illustrate how concurrent transactions work. The table shows the interleaved order in which the statements are performed by two transactions named T1 and T2. T1 retrieves rows, adds a row, and updates a row in the customers table. T2 retrieves rows from the customers table. T2 doesn’t see the changes made by T1 until T1 commits its changes. You can enter the statements shown in [Table 8-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch08.html#table_8-1) and see their results by starting two separate SQL\*Plus sessions and connecting as the store user for both sessions; you enter the statements in the interleaved order shown in the table into the SQL\*Plus sessions.

**Transaction Locking**

To support concurrent transactions, the Oracle database software must ensure that the data in the tables remains valid. It does this through the use of *locks*. Consider the following example in which two transactions named T1 and T2 attempt to modify customer #1 in the customers table:

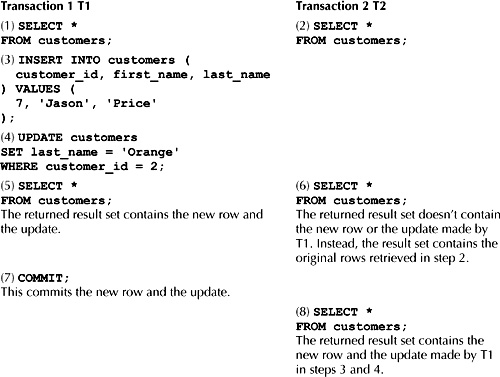
**1**. T1 performs an UPDATE to modify customer #1, but T1 doesn’t perform a COMMIT. T1 is said to have "locked" the row.

**2**. T2 also attempts to perform an UPDATE to modify customer #1, but since this row is already locked by T1, T2 is prevented from getting a lock on the row. T2’s UPDATE statement has to wait until T1 ends and frees the lock on the row.

**3**. T1 ends by performing a COMMIT, thus freeing the lock on the row.

**4**. T2 gets the lock on the row and the UPDATE is performed. T2 holds the lock on the row until T2 ends.

To summarize: A transaction cannot get a lock on a row while another transaction already holds the lock on that row.



**TABLE 8-1** *Concurrent Transactions*

Image

**NOTE**  
*The easiest way to understand default locking is as follows:* readers don’t block readers, writers don’t block readers, and writers only block writers when they attempt to modify the same row.

**Transaction Isolation Levels**

The *transaction isolation level* is the degree to which the changes made by one transaction are separated from other transactions running concurrently. Before you see the various transaction isolation levels available, you need to understand the types of problems that may occur when current transactions attempt to access the same rows in a table.

In the following list, you’ll see examples of two concurrent transactions named T1 and T2 that are accessing the same rows; listed are the three types of potential transaction processing problems:

Image **Phantom reads** T1 reads a set of rows returned by a specified WHERE clause. T2 then inserts a new row, which also happens to satisfy the WHERE clause of the query previously used by T1. T1 then reads the rows again using the same query, but now sees the additional row just inserted by T2. This new row is known as a "phantom" because to T1 this row seems to have magically appeared.

Image **Nonrepeatable reads** T1 reads a row, and T2 updates the same row just read by T1. T1 then reads the same row again and discovers that the row it read earlier is now different. This is known as a "nonrepeatable" read, because the row originally read by T1 has been changed.

Image **Dirty reads** T1 updates a row, but doesn’t commit the update. T2 then reads the updated row. T1 then performs a rollback, undoing the previous update. Now the row just read by T2 is no longer valid (it’s "dirty") because the update made by T1 wasn’t committed when the row was read by T2.

To deal with these potential problems, databases implement various levels of transaction isolation to prevent concurrent transactions from interfering with each other. The SQL standard defines the following transaction isolation levels, shown in order of increasing isolation:

Image **READ UNCOMMITTED** Phantom reads, nonrepeatable reads, and dirty reads are permitted.

Image **READ COMMITTED** Phantom reads and nonrepeatable reads are permitted, but dirty reads are not.

Image **REPEATABLE READ** Phantom reads are permitted, but nonrepeatable and dirty reads are not.

Image **SERIALIZABLE** Phantom reads, nonrepeatable reads, and dirty reads are not permitted.

The Oracle database software supports the READ COMMITTED and SERIALIZABLE transaction isolation levels. It doesn’t support READ UNCOMMITTED or REPEATABLE READ levels.

The default transaction isolation level defined by the SQL standard is SERIALIZABLE, but the default used by the Oracle database is READ COMMITTED, which is acceptable for nearly all applications.

Image

**CAUTION**  
*Although you can use* SERIALIZABLE *with the Oracle database, it may increase the time your SQL statements take to complete. You should only use* SERIALIZABLE *if you absolutely have to*.

You set the transaction isolation level using the SET TRANSACTION statement. For example, the following statement sets the transaction isolation level to SERIALIZABLE:

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

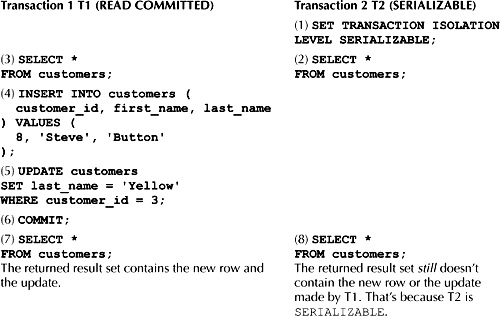
You’ll see an example of a transaction that uses the isolation level of SERIALIZABLE next.

**A SERIALIZABLE Transaction Example**

In this section, you’ll see an example that shows the effect of setting the transaction isolation level to SERIALIZABLE.

The example uses two transactions named T1 and T2. T1 has the default isolation level of READ COMMITTED; T2 has a transaction isolation level of SERIALIZABLE. T1 and T2 will read the rows in the customers table, and then T1 will insert a new row and update an existing row in the customers table. Because T2 is SERIALIZABLE, it doesn’t see the inserted row or the update made to the existing row made by T1, even *after*T1 commits its changes. That’s because reading the inserted row would be a phantom read, and reading the update would be a nonrepeatable read, which are not permitted by SERIALIZABLE transactions.

[Table 8-2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch08.html#table_8-2) shows the SQL statements that make up T1 and T2 in the interleaved order in which the statements are to be performed.



**TABLE 8-2** *SERIALIZABLE Transactions*

**QUERY FLASHBACKS**

If you mistakenly commit changes and you want to view rows as they originally were, you can use a query flashback. You can then use the results of a query flashback to manually change rows back to their original values if you need to.

Query flashbacks can be based on a datetime or system change number (SCN). The database uses SCNs to track changes made to data, and you can use them to flash back to a particular SCN in the database.

**Granting the Privilege for Using Flashbacks**

Flashbacks use the PL/SQL DBMS\_FLASHBACK package, for which you must have the EXECUTE privilege to run. The following example connects as the sys user and grants the EXECUTE privilege on DBMS\_FLASHBACKto the store user:

**CONNECT sys/change\_on\_install AS sysdba**

**GRANT EXECUTE ON SYS.DBMS\_FLASHBACK TO store;**

Image

**NOTE**  
*Speak with your DBA if you are unable to perform these statements. You’ll learn about privileges in the next chapter, and you’ll learn about PL/SQL packages in*[*Chapter 11*](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch11.html#ch11).

**Time Query Flashbacks**

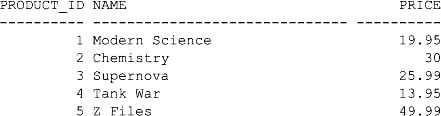
The following example connects as store and retrieves the product\_id, name, and price columns for the first five rows from the products table:

**CONNECT store/store\_password**

**SELECT product\_id, name, price**

**FROM products**

**WHERE product\_id <= 5;**



Image

**NOTE**  
*If you see different prices for any of these products, go ahead and rerun the* store\_schema.sql*file*.

The next example reduces the price of these rows, commits the change, and retrieves the rows again so you can see the new prices:

**UPDATE products**

**SET price = price \* 0.75**

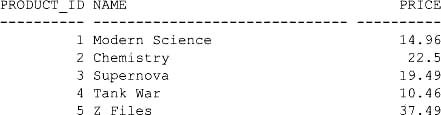
**WHERE product\_id <= 5;**

**COMMIT;**

**SELECT product\_id, name, price**

**FROM products**

**WHERE product\_id <= 5;**



The following statement executes the DBMS\_FLASHBACK.ENABLE\_AT\_TIME() procedure, which enables you to perform a flashback to a particular datetime; notice the DBMS\_FLASHBACK.ENABLE\_AT\_TIME()procedure accepts a datetime and the example passes SYSDATE − 10 / 1440 to the procedure (this expression evaluates to a datetime ten minutes in the past):

**EXECUTE DBMS\_FLASHBACK.ENABLE\_AT\_TIME(SYSDATE − 10 / 1440);**

Image

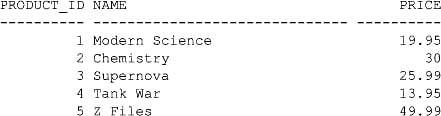
**NOTE**  
*24 hours × 60 minutes per hour = 1440 minutes. Therefore* SYSDATE – 10 / 1440 *is a datetime ten minutes in the past*.

Any queries you execute now will display the rows as they were ten minutes ago. Assuming you performed the earlier UPDATE less than ten minutes ago, the following query will display the prices as they were before you updated them:

**SELECT product\_id, name, price**

**FROM products**

**WHERE product\_id <= 5;**



To disable a flashback, you execute DBMS\_FLASHBACK.DISABLE(), as shown in the following example:

**EXECUTE DBMS\_FLASHBACK.DISABLE();**

Image

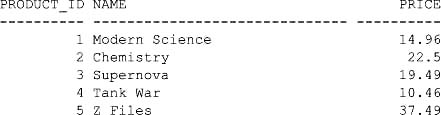
**CAUTION**  
*You must disable a flashback before you can enable it again*.

Now when you perform queries, the rows as they currently exist will be retrieved, as shown here:

**SELECT product\_id, name, price**

**FROM products**

**WHERE product\_id <= 5;**



**System Change Number Query Flashbacks**

Flashbacks based on system change numbers (SCNs) can be more precise than those based on a time, because the database uses SCNs to track changes made to data. To get the current SCN, you can execute DBMS\_FLASHBACK.GET\_SYSTEM\_CHANGE\_NUMBER(), as shown in the following example:

**VARIABLE current\_scn NUMBER**

**EXECUTE :current\_scn := DBMS\_FLASHBACK.GET\_SYSTEM\_CHANGE\_NUMBER();**

**PRINT current\_scn**

CURRENT\_SCN

-----------

292111

The next example adds a row to the products table, commits the change, and retrieves the new row:

**INSERT INTO products (**

**product\_id, product\_type\_id, name, description, price**

**) VALUES (**

**15, 1, 'Physics', 'Textbook on physics', 39.95**

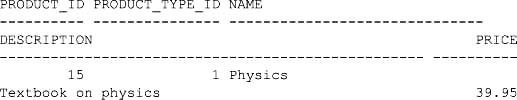
**);**

**COMMIT;**

**SELECT \***

**FROM products**

**WHERE product\_id = 15;**



The next example executes the following procedure, DBMS\_FLASHBACK.ENABLE\_AT\_SYSTEM\_CHANGE\_NUMBER(), which enables you to perform a flashback to an SCN; notice that this procedure accepts an SCN and that the example passes the current\_scn variable to the procedure:

**EXECUTE DBMS\_FLASHBACK.ENABLE\_AT\_SYSTEM\_CHANGE\_NUMBER(:current\_scn);**

Any queries you execute now will display the rows as they were at the SCN stored in current\_scn before you performed the INSERT. The following query attempts to get the row with a product\_id of 15; it fails because that new row was added after the SCN stored in current\_scn:

**SELECT product\_id**

**FROM products**

**WHERE product\_id = 15;**

no rows selected

To disable a flashback, you execute DBMS\_FLASHBACK.DISABLE(), as shown in the following example:

**EXECUTE DBMS\_FLASHBACK.DISABLE();**

If you perform the previous query again, you’ll see the new row that was added by the INSERT.

Image

**NOTE**  
*If you followed along with the examples, go ahead and rerun the* store\_schema.sql *script to recreate everything. That way, the results of your SQL statements will match mine as you progress through the rest of this book*.

**SUMMARY**

In this chapter, you have learned the following:

Image How to add rows using the INSERT statement.

Image How to modify rows using the UPDATE statement.

Image How to remove rows using the DELETE statement.

Image How the database maintains referential integrity through the enforcement of constraints.

Image How to use the DEFAULT keyword to specify default values for columns.

Image How to merge rows using the MERGE statement.

Image A database transaction is a group of SQL statements that comprise a logical unit of work.

Image The Oracle database software can handle multiple concurrent transactions.

Image How to use query flashbacks to view rows as they originally were before you made changes to them.

**CHAPTER 9  
Users, Privileges, and Roles**

In this chapter, you will do the following:

Image Learn more about users

Image See how privileges are used to enable users to perform tasks in the database

Image Explore the two types of privileges: system privileges and object privileges

Image Learn how system privileges allow you to perform actions such as executing DDL statements

Image See how object privileges allow you to perform actions such as executing DML statements

Image Explore how to group privileges together into roles

Image Learn how to audit the execution of SQL statements

Image

**NOTE**  
*You’ll need to type in the SQL statements shown in this chapter if you want to follow the examples: The statements are not contained in any script*.

**USERS**

In this section, you’ll learn how to create a user, alter a user’s password, and drop a user.

You will see the term "tablespace" used in this chapter. Tablespaces are used by the database to store separate objects, which can include tables, types, PL/SQL code, and so on. Typically, related objects are grouped together and stored in the same tablespace. For example, you might create an order entry application and store all the objects for that application in one tablespace, and you might create a supply chain application and store the objects for that application in a different tablespace. For more details on tablespaces, you should read the *Oracle Database Concepts* manual published by Oracle Corporation.

**Creating a User**

To create a user in the database, you use the CREATE USER statement. The simplified syntax for the CREATE USER statement is as follows:

CREATE USER *user\_name* IDENTIFIED BY *password*

[DEFAULT TABLESPACE *default\_tablespace*]

[TEMPORARY TABLESPACE *temporary\_tablespace*];

where

Image *user\_name* is the name of the database user.

Image *password* is the password for the database user.

Image *default\_tablespace* is the default tablespace where database objects are stored. If you omit a default tablespace, the default SYSTEM tablespace, which always exists in a database, is used.

Image *temporary\_tablespace* is the default tablespace where temporary objects are stored. These objects include temporary tables that you’ll learn about in the next chapter. If you omit a temporary tablespace, the default SYSTEM tablespace is used.

The following example connects as system and creates a user named jason with a password of price:

**CONNECT system/manager**

**CREATE USER jason IDENTIFIED BY price;**

Image

**NOTE**  
*If you want to follow along with these examples you’ll need to connect to the database as a privileged user. I used the* system *user in the example, which has a password of* manager *in my database*.

The next example creates a user named henry and specifies a default and temporary tablespace:

**CREATE USER henry IDENTIFIED BY hooray**

**DEFAULT TABLESPACE users**

**TEMPORARY TABLESPACE temp;**

Image

**NOTE**  
*If your database doesn’t have tablespaces named* users *and* temp, *you can skip this example. The*henry *user isn’t used elsewhere in this book, and I included the example only so you can see how to specify tablespaces for a user. You can view all the tablespaces in a database by connecting as the*system *user and running the query* SELECT tablespace\_name FROM dba\_tablespaces.

If you want a user to be able to do things in the database, that user must be granted the necessary permissions to do those things. For example, to connect to the database a user must be granted the permission to create a session, which is the CREATE SESSION system privilege. Permissions are granted by a privileged user (system, for example) using the GRANT statement.

The following example grants the CREATE SESSION permission to jason:

**GRANT CREATE SESSION TO jason;**

The jason user will now be able to connect to the database.

The following example creates other users that are used in this chapter and grants the CREATE SESSIONprivilege to those users:

**CREATE USER steve IDENTIFIED BY button;**

**CREATE USER gail IDENTIFIED BY seymour;**

**GRANT CREATE SESSION TO steve, gail;**

**Changing a User’s Password**

You can change a user’s password using the ALTER USER statement. For example, the following statement changes the password for jason to marcus:

**ALTER USER jason IDENTIFIED BY marcus;**

You can also change the password for the user you’re currently logged in as using the PASSWORD command. After you enter PASSWORD, SQL\*Plus prompts you to enter the old password and the new password twice for confirmation. The following example connects as jason and executes PASSWORD; notice the password itself is masked using asterisks:

**CONNECT jason/marcus**

**PASSWORD**

Changing password for JASON

Old password: \*\*\*\*\*\*

New password: \*\*\*\*\*\*

Retype new password: \*\*\*\*\*\*

Password changed

**Deleting a User**

You delete a user using the DROP USER statement. The following example connects as system and uses DROP USER to delete jason:

**CONNECT system/manager**

**DROP USER jason;**

Image

**NOTE**  
*You must add the keyword* CASCADE *after the user’s name in the* DROP USER *statement if that user’s schema contains objects such as tables. However, you should ensure no other users need access to those objects before doing this*.

**SYSTEM PRIVILEGES**

A *system privilege* allows a user to perform certain actions within the database, such as executing DDL statements. For example, CREATE TABLE allows a user to create a table in their schema. Some of the commonly used system privileges are shown in [Table 9-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch09.html#table_9-1).

Image

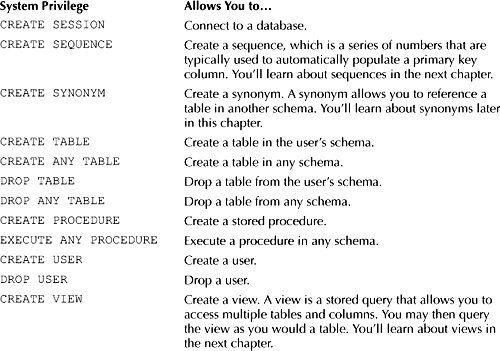
**NOTE**  
*You can get the full list of system privileges in the* Oracle Database SQL Reference *manual published by Oracle Corporation*.

As you’ll see later, privileges can be grouped together into *roles*. Two useful roles to grant to a user are CONNECT and RESOURCE; CONNECT allows a user to connect to the database; RESOURCE allows a user to create various database objects like tables, sequences, PL/SQL code, and so on.

**Granting System Privileges to a User**

You use GRANT to grant a system privilege to a user. The following example grants some system privileges to steve (assuming you’re still connected to the database as system):

**GRANT CREATE SESSION, CREATE USER, CREATE TABLE TO steve;**



**TABLE 9-1** *Commonly Used System Privileges*

You can also use WITH ADMIN OPTION to allow a user to grant a privilege to another user. The following example grants the EXECUTE ANY PROCEDURE privilege with the ADMIN option to steve:

**GRANT EXECUTE ANY PROCEDURE TO steve WITH ADMIN OPTION;**

EXECUTE ANY PROCEDURE can then be granted to another user by steve. The following example connects as steve and grants EXECUTE ANY PROCEDURE to gail:

**CONNECT steve/button**

**GRANT EXECUTE ANY PROCEDURE TO gail;**

You can grant a privilege to all users by granting to PUBLIC. The following example connects as system and grants EXECUTE ANY PROCEDURE to PUBLIC:

**CONNECT system/manager**

**GRANT EXECUTE ANY PROCEDURE TO PUBLIC;**

Every user in the database now has the EXECUTE ANY PROCEDURE privilege.

**Checking System Privileges Granted to a User**

You can check which system privileges a user has by querying user\_sys\_privs. [Table 9-2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch09.html#table_9-2) describes some of the columns in user\_sys\_privs.

Image

**NOTE**  
user\_sys\_privs *forms part of the Oracle database’s data dictionary. The data dictionary stores information about the database itself*.

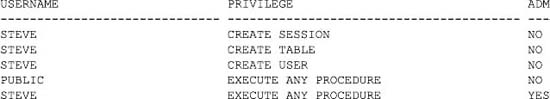
The following example connects as steve and queries user\_sys\_privs:

**CONNECT steve/button**

**SELECT \***

**FROM user\_sys\_privs**

**ORDER BY privilege;**



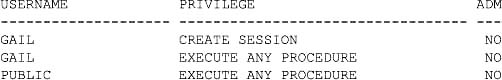
The next example connects as gail and queries user\_sys\_privs:

**CONNECT gail/seymour**

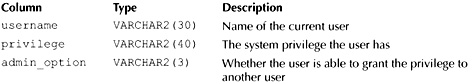
**SELECT \***

**FROM user\_sys\_privs**

**ORDER BY privilege;**



Notice gail has the EXECUTE ANY PROCEDURE privilege that was granted earlier by steve.



**TABLE 9-2** *Some Columns in user\_sys\_privs*

**Making Use of System Privileges**

Once a user has been granted a system privilege, they can use it to perform the specified task. For example, steve has the CREATE USER privilege, so he is able to create a user:

**CONNECT steve/button**

**CREATE USER roy IDENTIFIED BY williams;**

If steve were to attempt to use a system privilege he doesn’t have, the database will return the error ORA-01031: insufficient privileges. For example, steve doesn’t have the DROP USER privilege, and in the following example steve attempts to drop roy and fails:

SQL> **DROP USER roy;**

DROP USER roy

\*

ERROR at line 1:

ORA-01031: insufficient privileges

**Revoking System Privileges from a User**

You revoke system privileges from a user using REVOKE. The following example connects as system and revokes the CREATE TABLE privilege from steve:

**CONNECT system/manager**

**REVOKE CREATE TABLE FROM steve;**

The next example revokes EXECUTE ANY PROCEDURE from steve:

**REVOKE EXECUTE ANY PROCEDURE FROM steve;**

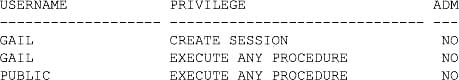
When you revoke EXECUTE ANY PROCEDURE from steve—who has already passed on this privilege to gail—then gail still keeps the privilege:

**CONNECT gail/seymour**

**SELECT \***

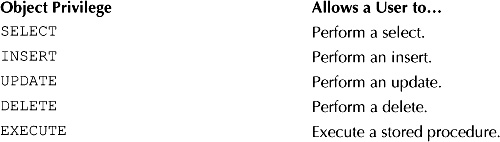
**FROM user\_sys\_privs**

**ORDER BY privilege;**



**OBJECT PRIVILEGES**

An *object privilege* allows a user to perform certain actions on database objects, such as executing DML statements on tables. For example, INSERT ON store.products allows a user to insert rows into the products table of the store schema. Some of the commonly used object privileges are shown in [Table 9-3](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch09.html#table_9-3).



**TABLE 9-3** *Commonly Used Object Privileges*

Image

**NOTE**  
*You can get the full list of system privileges in the* Oracle Database SQL Reference *manual published by Oracle Corporation*.

**Granting Object Privileges to a User**

You use GRANT to grant an object privilege to a user. The following example connects as store and grants the SELECT, INSERT, and UPDATE object privileges on the products table to steve along with the SELECTprivilege on the employees table:

**CONNECT store/store\_password**

**GRANT SELECT, INSERT, UPDATE ON store.products TO steve;**

**GRANT SELECT ON store.employees TO steve;**

The next example grants the UPDATE privilege on the last\_name and salary columns to steve:

**GRANT UPDATE (last\_name, salary) ON store.employees TO steve;**

You can also use the GRANT option to enable a user to grant a privilege to another user. The following example grants the SELECT privilege on the customers table with the GRANT option to steve:

**GRANT SELECT ON store.customers TO steve WITH GRANT OPTION;**

Image

**NOTE**  
*You use the* GRANT *option to allow a user to grant an* object privilege *to another user, and you use the*ADMIN *option to allow a user to grant a* system privilege *to another user*.

The SELECT ON store.customers privilege can then be granted to another user by steve. The following example connects as steve and grants this privilege to gail:

**CONNECT steve/button**

**GRANT SELECT ON store.customers TO gail;**

**Checking Object Privileges Made**

You can check which table object privileges a user has made to other users by querying user\_tab\_privs\_made. [Table 9-4](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch09.html#table_9-4) documents the columns in user\_tab\_privs\_made.

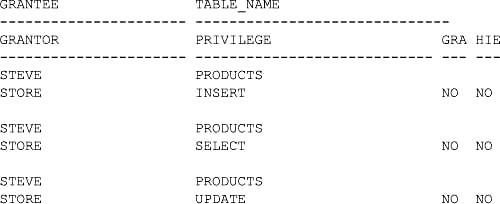
The following example connects as store and queries user\_tab\_privs\_made. Because there are so many rows, I’ll limit the retrieved rows to those where table\_name is PRODUCTS:

**CONNECT store/store\_password**

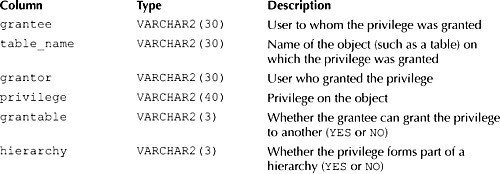
**SELECT \***

**FROM user\_tab\_privs\_made**

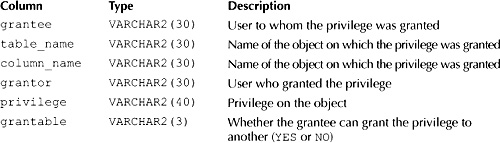
**WHERE table\_name = 'PRODUCTS' ;**



You can check which column object privileges a user has made by querying user\_col\_privs\_made. [Table 9-5](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch09.html#table_9-5) documents the columns in user\_col\_privs\_made.



**TABLE 9-4** *Some Columns in user\_tab\_privs\_made*

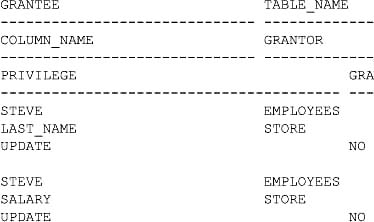


**TABLE 9-5** *Some Columns in user\_col\_privs\_made*

The following example queries user\_col\_privs\_made:

**SELECT \***

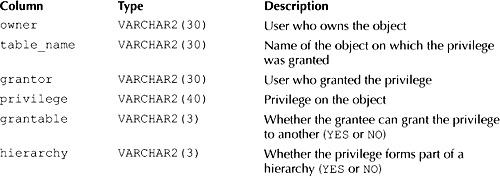
**FROM user\_col\_privs\_made;**



**Checking Object Privileges Received**

You can check which object privileges on a table a user has received by querying the user\_tab\_privs\_recd table. [Table 9-6](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch09.html#table_9-6) documents the columns in user\_tab\_privs\_recd.

The next example connects as steve and queries user\_tab\_privs\_recd:



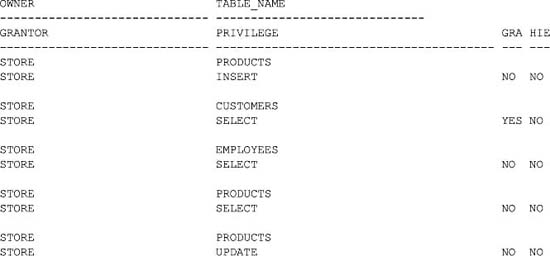
**TABLE 9-6** *Some Columns in user\_tab\_privs\_recd*

**CONNECT steve/button**

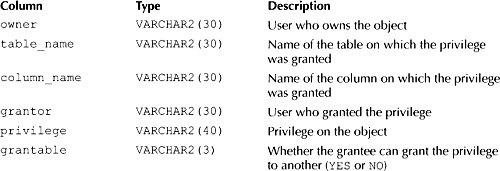
**SELECT \***

**FROM user\_tab\_privs\_recd**

**ORDER BY privilege;**



You can check which column object privileges a user has received by querying user\_col\_privs\_recd. [Table 9-7](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch09.html#table_9-7) documents the columns in user\_col\_privs\_recd.

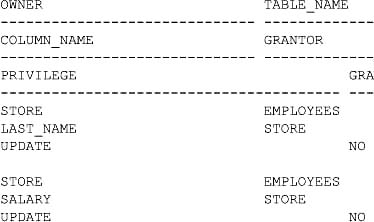


**TABLE 9-7** *Some Columns in user\_col\_privs\_recd*

The following example queries user\_col\_privs\_recd:

**SELECT \***

**FROM user\_col\_privs\_recd;**



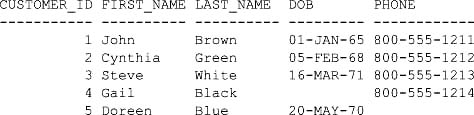
**Making Use of Object Privileges**

Once a user has been granted an object privilege, they can use it to perform the specified task. For example, steve has the SELECT privilege on store.customers:

**CONNECT steve/button**

**SELECT \***

**FROM store.customers;**



If steve were to attempt to retrieve from the purchases table—for which he doesn’t have any permissions—the database will return the error ORA-00942: table or view does not exist:

SQL> **SELECT \***

2 **FROM store.purchases;**

FROM store.purchases

\*

ERROR at line 2:

ORA-00942: table or view does not exist

**Synonyms**

In the examples in the previous section, you saw that you can access tables in another schema by specifying the schema name followed by the table. For example, when steve retrieved rows from the customers table in the store schema, he performed a query on store.customers. You can avoid having to enter the schema name by creating a *synonym* for a table, which you do by using the CREATE SYNONYM statement.

Let’s take a look at an example. First, connect as system and grant the CREATE SYNONYM system privilege to steve:

**CONNECT system/manager**

**GRANT CREATE SYNONYM TO steve;**

Next, connect as steve and perform a CREATE SYNONYM statement to create a synonym for the store.customers table:

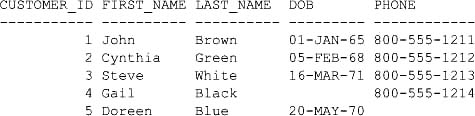
**CONNECT steve/button**

**CREATE SYNONYM customers FOR store.customers;**

To retrieve rows from store.customers, all steve has to do is to reference the customers synonym in the FROM clause of a SELECT statement. For example:

**SELECT \***

**FROM customers;**



**Public Synonyms**

You can also create a *public* synonym for a table. When you do this, all users see the synonym. The following tasks

Image Connect as system

Image Grant the CREATE PUBLIC SYNONYM system privilege to store

Image Connect as store

Image Create a public synonym named products for store.products

are performed by the following statements:

**CONNECT system/manager**

**GRANT CREATE PUBLIC SYNONYM TO store;**

**CONNECT store/store\_password**

**CREATE PUBLIC SYNONYM products FOR store.products;**

If you connect as steve, who has the SELECT privilege on store.products, you can now retrieve rows from store.products through the products public synonym:

**CONNECT steve/button**

**SELECT \***

**FROM products;**

Even though a public synonym has been created for store.products, a user still needs object privileges on that table to actually access the table. For example, gail can see the products public synonym, but gaildoesn’t have any object privileges on store.products. Therefore, if gail attempts to retrieve rows from products, the database returns the error ORA-00942:table or view does not exist:

SQL> **CONNECT gail/seymour**

Connected.

SQL> **SELECT \* FROM products;**

SELECT \* FROM products

\*

ERROR at line 1:

ORA-00942: table or view does not exist

If gail had the SELECT object privilege on the store.products table, the previous SELECT would succeed.

If a user has other object privileges, that user can exercise those object privileges through a synonym. For example, if gail had the INSERT object privilege on the store.products table, gail would be able to add a row to store.products through the products synonym.

**Revoking Object Privileges**

You revoke object privileges using REVOKE. The following example connects as store and revokes the INSERT privilege on the products table from steve:

**CONNECT store/store\_password**

**REVOKE INSERT ON products FROM steve;**

The next example revokes the SELECT privilege on the customers table from steve:

**REVOKE SELECT ON store.customers FROM steve;**

When you revoke SELECT ON store.customers from steve—who has already passed on this privilege to gail—gail also loses the privilege.

**ROLES**

A *role* is a group of privileges that you can assign to a user or to another role. The following points summarize the benefits and features of roles:

Image Rather than assigning privileges one at a time directly to a user, you can create a role, assign privileges to that role, and then grant that role to multiple users and roles.

Image When you add or delete a privilege from a role, all users and roles assigned that role automatically receive or lose that privilege.

Image You can assign multiple roles to a user or role.

Image You can assign a password to a role.

As you can see from these points, roles can help you manage multiple privileges assigned to multiple users.

**Creating Roles**

To create a role, you must have the CREATE ROLE system privilege. As you’ll see in a later example, the store user also needs the ability to grant the CREATE USER system privilege with the ADMIN option. The following example connects as system and grants the required privileges to store:

**CONNECT system/manager**

**GRANT CREATE ROLE TO store;**

**GRANT CREATE USER TO store WITH ADMIN OPTION;**

[Table 9-8](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch09.html#table_9-8) shows the roles you’ll create shortly.

You create a role using the CREATE ROLE statement. The following statements connect as store and create the three roles shown in [Table 9-8](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch09.html#table_9-8):

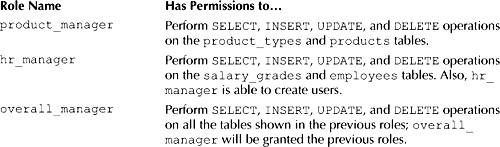
**CONNECT store/store\_password**

**CREATE ROLE product\_manager;**

**CREATE ROLE hr\_manager;**

**CREATE ROLE overall\_manager IDENTIFIED by manager\_password;**

Notice overall\_manager has a password of manager\_password.



**TABLE 9-8** *Roles to Be Created*

**Granting Privileges to Roles**

You grant privileges to a role using the GRANT statement. You can grant both system and object privileges to a role as well as grant another role to a role. The following example grants the required privileges to the product\_manager and hr\_manager roles and grants these two roles to overall\_manager:

**GRANT SELECT, INSERT, UPDATE, DELETE ON product\_types TO product\_manager;**

**GRANT SELECT, INSERT, UPDATE, DELETE ON products TO product\_manager;**

**GRANT SELECT, INSERT, UPDATE, DELETE ON salary\_grades TO hr\_manager;**

**GRANT SELECT, INSERT, UPDATE, DELETE ON employees TO hr\_manager;**

**GRANT CREATE USER TO hr\_manager;**

**GRANT product\_manager, hr\_manager TO overall\_manager;**

**Granting Roles to a User**

You grant a role to a user using GRANT. The following example grants the overall\_manager role to steve:

**GRANT overall\_manager TO steve;**

**Checking Roles Granted to a User**

You can check which roles have been granted to a user by querying user\_role\_privs. [Table 9-9](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch09.html#table_9-9) defines the columns in user\_role\_privs.

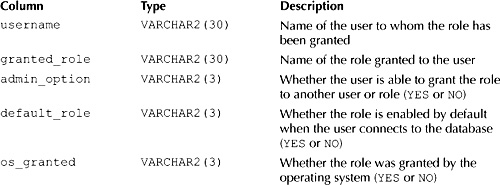
The following example connects as steve and queries user\_role\_privs:

**CONNECT steve/button**

**SELECT \***

**FROM user\_role\_privs;**

Image



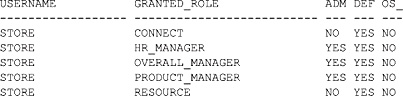
**TABLE 9-9** *Some Columns in user\_role\_privs*

A user who creates a role is also granted that role by default. The following example connects as store and queries user\_role\_privs:

**CONNECT store/store\_password**

**SELECT \***

**FROM user\_role\_privs;**



Notice store has the roles CONNECT and RESOURCE in addition to the roles store created earlier.

Image

**NOTE**  
CONNECT *and* RESOURCE *are built-in roles that were granted to* store *when you ran the*store\_schema.sql *script. As you’ll see in the next section, the* CONNECT *and* RESOURCE *roles contain multiple privileges*.

**Checking System Privileges Granted to a Role**

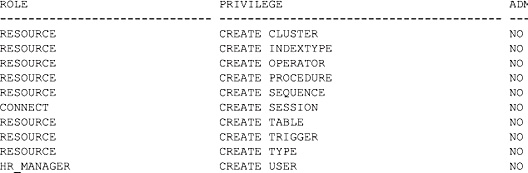
You can check which system privileges have been granted to a role by querying role\_sys\_privs. [Table 9-10](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch09.html#table_9-10)defines the columns in role\_sys\_privs.

The following example retrieves the rows from role\_sys\_privs (assuming you’re still connected as store):

**SELECT \***

**FROM role\_sys\_privs**

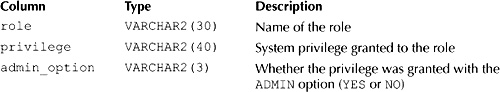
**ORDER BY privilege;**



Notice that the RESOURCE role has many privileges assigned to it.

Image

**NOTE**  
*The previous query was run using Oracle Database* 11*g. If you are using a different version of the database software, you may get slightly different results*.



**TABLE 9-10** *Some Columns in role\_sys\_privs*

**Checking Object Privileges Granted to a Role**

You can check which object privileges have been granted to a role by querying role\_tab\_privs. [Table 9-11](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch09.html#table_9-11)defines the columns in role\_tab\_privs.

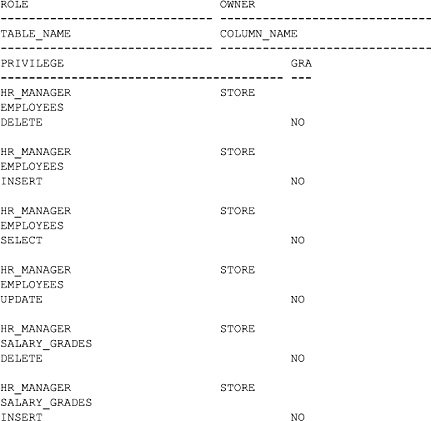
The following example queries role\_tab\_privs where role equals HR\_MANAGER:

**SELECT \***

**FROM role\_tab\_privs**

**WHERE role='HR\_MANAGER'**

**ORDER BY table\_name;**





**Making Use of Privileges Granted to a Role**

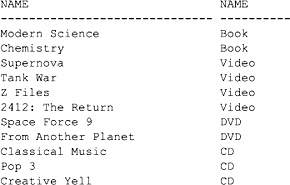
Once a user has been granted a privilege via a role, they can use that privilege to perform the authorized tasks. For example, steve has the overall\_manager role. The overall\_manager was granted the product\_manager and hr\_manager roles. The product\_manager was granted the SELECT object privilege on the products and product\_types tables. Therefore, steve is able to retrieve rows from these tables, as shown in the following example:

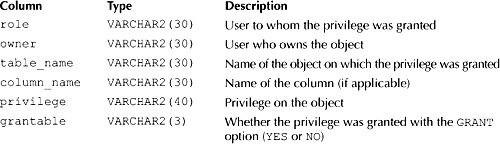
**CONNECT steve/button**

**SELECT p.name, pt.name**

**FROM store.products p, store.product\_types pt**

**WHERE p.product\_type\_id = pt.product\_type\_id;**





**TABLE 9-11** *Some Columns in role\_tab\_privs*

**Default Roles**

By default, when a role is granted to a user, that role is enabled for that user. This means that when the user connects to the database, the role is automatically available to them. To enhance security, you can disable a role by default; when the user connects, they will have to enable the role themselves before they can use it. If the role has a password, the user must enter that password before the role is enabled. For example, the overall\_manager role has a password of manager\_passsword, and overall\_manager is granted to steve. In the example you’ll see next, you’ll disable overall\_manager so that steve has to enable this role and enter the password before he can use it. You do this by altering a role so that it is no longer a default role using the ALTER ROLE statement. The following example connects as system and alters steve so that overall\_manager is no longer a default role:

**CONNECT system/manager**

**ALTER USER steve DEFAULT ROLE ALL EXCEPT overall\_manager;**

When you connect as steve, you need to enable overall\_manager using SET ROLE:

**CONNECT steve/button**

**SET ROLE overall\_manager IDENTIFIED BY manager\_password;**

Once you’ve set the role, you can use the privileges granted to that role. You can set your role to "none" (i.e. no role) using the following statement:

**SET ROLE NONE;**

You can also set your role to "all roles" except overall\_manager using the following statement:

**SET ROLE ALL EXCEPT overall\_manager;**

By assigning passwords to roles and setting roles to not be enabled by default for a user, you introduce an additional level of security.

**Revoking a Role**

You revoke a role using REVOKE. The following example connects as store and revokes the overall\_manager role from steve:

**CONNECT store/store\_password**

**REVOKE overall\_manager FROM steve;**

**Revoking Privileges from a Role**

You revoke a privilege from a role using REVOKE. The following example connects as store and revokes all privileges on the products and product\_types tables from product\_manager (assuming you’re still connected as store):

**REVOKE ALL ON products FROM product\_manager;**

**REVOKE ALL ON product\_types FROM product\_manager;**

**Dropping a Role**

You drop a role using DROP ROLE. The following example drops the overall\_manager, product\_manager, and hr\_manager roles (assuming you’re still connected as store):

**DROP ROLE overall\_manager;**

**DROP ROLE product\_manager;**

**DROP ROLE hr\_manager;**

**AUDITING**

The Oracle database software contains auditing capabilities that enable you to keep track of database operations. Some operations may be audited at a high level, such as failed attempts to log into the database, while others may be audited at a detailed level, such as when a user retrieved rows from a specific table. Typically, your database administrator will be responsible for enabling auditing and monitoring the output for security violations. In this section, you will see some simple examples of auditing, which is performed using the AUDIT statement.

**Privileges Required to Perform Auditing**

Before a user can issue AUDIT statements, that user must have been granted certain privileges:

Image For auditing high-level operations, the user must have the AUDIT SYSTEM privilege. An example of a high-level operation is the issuance of *any* SELECT statement, regardless of the table involved.

Image For tracking operations on specific database objects, the user must either have the AUDIT ANY privilege or the database object must be in their schema. An example of specific database object operation is the issuance of a SELECT statement for a particular table.

The following example connects to the database as the system user and grants the AUDIT SYSTEM and AUDIT ANY privileges to the store user:

**CONNECT system/manager**

**GRANT AUDIT SYSTEM TO store;**

**GRANT AUDIT ANY TO store;**

**Auditing Examples**

The following example connects to the database as the store user and audits the issuance of CREATE TABLEstatements:

**CONNECT store/store\_password**

**AUDIT CREATE TABLE;**

As a result of this AUDIT statement, any CREATE TABLE statements issued will be audited; for example, the following statement creates a simple test table:

**CREATE TABLE test (**

**id INTEGER**

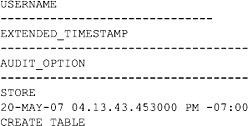
**);**

You can view the audit trail of information for the user you are currently logged in as through the USER\_AUDIT\_TRAIL view. The following example shows the audit record generated by the previous CREATE TABLE statement:

**SELECT username, extended\_timestamp, audit\_option**

**FROM user\_audit\_trail**

**WHERE audit\_option='CREATE TABLE';**



You may also audit the issuance of statements by a particular user. The following example audits all SELECTstatements issued by the store user:

**AUDIT SELECT TABLE BY store;**

The next example audits all INSERT, UPDATE, and DELETE statements made by the store and steve users:

**AUDIT INSERT TABLE, UPDATE TABLE, DELETE TABLE BY store, steve;**

You may also audit the issuance of statements made for a particular database object. The following example audits all SELECT statements issued for the products table:

**AUDIT SELECT ON store.products;**

The next example audits all statements issued for the employees table:

**AUDIT ALL ON store.employees;**

You may also use the WHENEVER SUCCESSFUL and WHENEVER NOT SUCCESSFUL options to indicate when auditing should be performed. WHENEVER SUCCESSFUL indicates auditing will be performed when the statement executed successfully. WHENEVER NOT SUCCESSFUL indicates auditing will be performed when the statement did not execute successfully. The default is to do both, that is, audit regardless of success. The following examples use the WHENEVER NOT SUCCESSFUL option:

**AUDIT UPDATE TABLE BY steve WHENEVER NOT SUCCESSFUL;**

**AUDIT INSERT TABLE WHENEVER NOT SUCCESSFUL;**

The next example uses the WHENEVER SUCCESSFUL option to audit the creation and deletion of a user:

**AUDIT CREATE USER, DROP USER WHENEVER SUCCESSFUL;**

The next example uses the WHENEVER SUCCESSFUL option to audit the creation and deletion of a user by the store user:

**AUDIT CREATE USER, DROP USER BY store WHENEVER SUCCESSFUL;**

You may also use the BY SESSION and BY ACCESS options. The BY SESSION option causes only one audit record to be logged when the same type of statement is issued during the same user database session; a database session starts when the user logs into the database and ends when the user logs out. The BY ACCESS option causes one audit record to be logged every time the same type of statement is issued, regardless of the user session. The following examples show the use of the BY SESSION and BY ACCESS options:

**AUDIT SELECT ON store.products BY SESSION;**

**AUDIT DELETE ON store.employees BY ACCESS;**

**AUDIT INSERT, UPDATE ON store.employees BY ACCESS;**

**Audit Trail Views**

Earlier, you saw the use of the USER\_AUDIT\_TRAIL view. This and the other audit trail views are outlined in the following list:

Image **USER\_AUDIT\_OBJECT** displays the audit records for all objects accessible to the current user.

Image **USER\_AUDIT\_SESSION** displays the audit records for connections and disconnections of the current user.

Image **USER\_AUDIT\_STATEMENT** displays the audit records for GRANT, REVOKE, AUDIT, NOAUDIT, and ALTER SYSTEM statements issued by the current user.

Image **USER\_AUDIT\_TRAIL** displays all audit trail entries related to the current user.

You may use these views to examine the contents of the audit trail. There are a number of similarly named views that the database administrator may use to examine the audit trail; these views are named DBA\_AUDIT\_OBJECT, DBA\_AUDIT\_SESSION, DBA\_AUDIT\_STATEMENT, DBA\_AUDIT\_TRAIL, plus others. These views allow the DBA to view audit records across all users. For more details on these views, you should consult the *Oracle Database Reference* manual published by Oracle Corporation.

**SUMMARY**

In this chapter, you’ve learned the following:

Image A user is created using the CREATE USER statement.

Image System privileges allow you to perform certain actions within the database, such as executing DDL statements.

Image Object privileges allow you to perform certain actions on database objects, such as executing DML statements on tables.

Image You can avoid having to enter the schema name by creating a synonym for a table.

Image A role is a group of privileges that you can assign to a user or another role.

Image Auditing the execution of SQL statements can be performed using the AUDIT statement.

**CHAPTER 10  
Creating Tables, Sequences, Indexes, and Views**

In this chapter, you will do the following:

Image Learn more about tables

Image See how to create and use sequences, which generate a series of numbers

Image Explore how to create and use indexes, which can improve the performance of queries

Image Learn how to create and use views, which are predefined queries that allow you to hide complexity from users, among other benefits

Image Examine flashback data archives, new for Oracle Database 11*g*, which store changes made to a table over a period of time

Let’s plunge in and examine tables.

**TABLES**

In this section, you’ll learn more about creating a table. You’ll see how to modify and drop a table as well as how to retrieve information about a table from the data dictionary. The data dictionary contains information about all the database items, such as tables, sequences, indexes, and so on.

**Creating a Table**

You use the CREATE TABLE statement to create a table. The simplified syntax for the CREATE TABLEstatement is as follows:

CREATE [GLOBAL TEMPORARY] TABLE *table\_name* (

*column\_name type* [CONSTRAINT *constraint\_def* DEFAULT *default\_exp*]

[, *column\_name type* [CONSTRAINT *constraint\_def* DEFAULT *default\_exp*] …]

)

[ON COMMIT {DELETE | PRESERVE} ROWS]

TABLESPACE *tab\_space*;

where

Image GLOBAL TEMPORARY means the table’s rows are temporary (these tables are known as temporary tables). The rows in a temporary table are specific to a user session, and how long the rows persist is set in the ON COMMIT clause.

Image *table\_name* is the name of the table.

Image *column\_name* is the name of a column.

Image *type* is the type of a column.

Image *constraint\_def* is a constraint on a column.

Image *default\_exp* is an expression to assign a default value to a column.

Image ON COMMIT controls the duration of the rows in a temporary table. DELETE means the rows are deleted at the end of a transaction. PRESERVE means the rows are kept until the end of a user session, at which point the rows are deleted. If you omit ON COMMIT for a temporary table, then the default DELETE is used.

Image *tab\_space* is the tablespace for the table. If you omit a tablespace, then the table is stored in the user’s default tablespace.

Image

**NOTE**  
*The full* CREATE TABLE *syntax is far richer than that shown above. For full details, see the* Oracle Database SQL Reference *book published by Oracle Corporation*.

The following example connects as the store user and creates a table named order\_status2:

**CONNECT store/store\_password**

**CREATE TABLE order\_status2 (**

**id INTEGER CONSTRAINT order\_status2\_pk PRIMARY KEY,**

**status VARCHAR2(10),**

**last\_modified DATE DEFAULT SYSDATE**

**);**

Image

**NOTE**  
*If you want to follow along with the examples in this chapter, you’ll need to enter and run the SQL statements using SQL\*Plus*.

The next example creates a temporary table named order\_status\_temp whose rows will be kept until the end of a user session (ON COMMIT PRESERVE ROWS):

**CREATE GLOBAL TEMPORARY TABLE order\_status\_temp (**

**id INTEGER,**

**status VARCHAR2(10),**

**last\_modified DATE DEFAULT SYSDATE**

**)**

**ON COMMIT PRESERVE ROWS;**

The next example performs the following:

Image Adds a row to order\_status\_temp.

Image Disconnects from the database to end the session, which causes the row in order\_status\_temp to be deleted.

Image Reconnects as store and queries order\_status\_temp, which shows there are no rows in this table.

**INSERT INTO order\_status\_temp (**

**id, status**

**) VALUES (**

**1, 'New'**

**);**

1 row created.

**DISCONNECT**

**CONNECT store/store\_password**

**SELECT \***

**FROM order\_status\_temp;**

no rows selected

**Getting Information on Tables**

You can get information about your tables by

Image Performing a DESCRIBE command on the table. You’ve already seen examples that use the DESCRIBEcommand in earlier chapters.

Image Querying the user\_tables view, which forms part of the data dictionary.

[Table 10-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch10.html#table_10-1) describes some of the columns in the user\_tables view.

Image

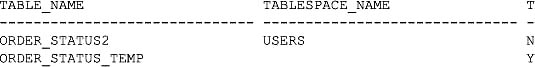
**NOTE**  
*You can retrieve information on all the tables you have access to by querying the* all\_tables *view*.

The following example retrieves some of the columns from user\_tables where the table\_name is order\_status2 or order\_status\_temp:

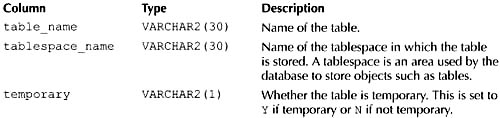
**SELECT table\_name, tablespace\_name, temporary**

**FROM user\_tables**

**WHERE table\_name IN ('ORDER\_STATUS2', 'ORDER\_STATUS\_TEMP');**



Notice the order\_status\_temp table is temporary, as indicated by the Y in the last column.



**TABLE 10-1** *Some Columns in the user\_tables View*

**Getting Information on Columns in Tables**

You can retrieve information about the columns in your tables from the user\_tab\_columns view. [Table 10-2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch10.html#table_10-2)describes some of the columns in user\_tab\_columns.

Image

**NOTE**  
*You can retrieve information on all the columns in tables you have access to by querying the*all\_tab\_columns *view*.

The following example retrieves some of the columns from user\_tab\_columns for the products table:

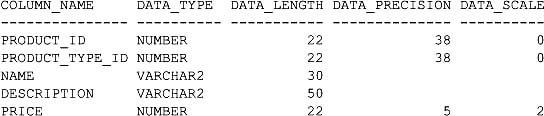
**COLUMN column\_name FORMAT a15**

**COLUMN data\_type FORMAT a10**

**SELECT column\_name, data\_type, data\_length, data\_precision, data\_scale**

**FROM user\_tab\_columns**

**WHERE table\_name = 'PRODUCTS';**



**Altering a Table**

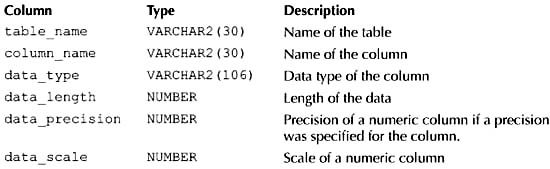
You alter a table using the ALTER TABLE statement. You can use ALTER TABLE to perform tasks such as

Image Adding, modifying, or dropping a column

Image Adding or dropping a constraint

Image Enabling or disabling a constraint

In the following sections, you’ll learn how to use ALTER TABLE to perform each of these tasks.



**TABLE 10-2** *Some Columns in the user\_tab\_columns View*

**Adding a Column**

The following example uses ALTER TABLE to add an INTEGER column named modified\_by to the order\_status2 table:

**ALTER TABLE order\_status2**

**ADD modified\_by INTEGER;**

The next example adds a column named initially\_created to order\_status2:

**ALTER TABLE order\_status2**

**ADD initially\_created DATE DEFAULT SYSDATE NOT NULL;**

You can verify the addition of the new column by executing a DESCRIBE command on order\_status2:

**DESCRIBE order\_status2**



**Adding a Virtual Column**

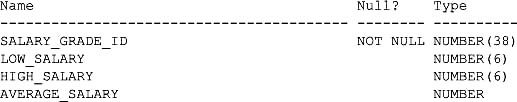
In Oracle Database 11*g*, you can add a virtual column, which is a column that refers only to other columns already in the table. For example, the following ALTER TABLE statement adds a virtual column named average\_salary to the salary\_grades table:

**ALTER TABLE salary\_grades**

**ADD (average\_salary AS ((low\_salary + high\_salary)/2));**

Notice average\_salary is set to the average of the low\_salary and high\_salary values. The following DESCRIBE command confirms the addition of the average\_salary column to the salary\_grades table:

**DESCRIBE salary\_grade s**



The following query retrieves the rows from the salary\_grades table:

**SELECT \***

**FROM salary\_grades;**

SALARY\_GRADE\_ID LOW\_SALARY HIGH\_SALARY AVERAGE\_SALARY

--------------- ---------- ----------- --------------

1 1 250000 125000.5

2 250001 500000 375000.5

3 500001 750000 625000.5

4 750001 999999 875000

**Modifying a Column**

The following list shows some of the column aspects you can modify using ALTER TABLE:

Image Change the size of a column (if the data type is one whose length may be changed, such as CHAR or VARCHAR2)

Image Change the precision of a numeric column

Image Change the data type of a column

Image Change the default value of a column

You’ll see examples of how to change these column aspects in the following sections.

**Changing the Size of a Column**

The following ALTER TABLE statement increases the maximum length of the order\_status2 .statuscolumn to 15 characters:

**ALTER TABLE order\_status2**

**MODIFY status VARCHAR2(15);**

Image

**CAUTION**  
*You can only* decrease *the length of a column if there are no rows in the table or all the rows contain null values for that column*.

**Changing the Precision of a Numeric Column**

The following ALTER TABLE statement changes the precision of the order\_status2.id column to 5:

**ALTER TABLE order\_status2**

**MODIFY id NUMBER(5);**

Image

**CAUTION**  
*You can only* decrease *the precision of a numeric column if there are no rows in the table or the column contains null values*.

**Changing the Data Type of a Column**

The following ALTER TABLE statement changes the data type of the order\_status2.status column to CHAR:

**ALTER TABLE order\_status2**

**MODIFY status CHAR(15);**

If the table is empty or the column contains null values, you can change the column to any data type (including a data type that is shorter); otherwise, you can change the data type of a column only to a compatible data type. For example, you can change a VARCHAR2 to CHAR (and vice versa) as long as you don’t make the column shorter; you cannot change a DATE to a NUMBER.

**Changing the Default Value of a Column**

The following ALTER TABLE statement changes the default value for the order\_status2.last\_modified column to SYSDATE - 1:

**ALTER TABLE order\_status2**

**MODIFY last\_modified DEFAULT SYSDATE - 1;**

The default value applies only to new rows added to the table. New rows will get their last\_ modifiedcolumn set to the current date minus one day.

**Dropping a Column**

The following ALTER TABLE statement drops the order\_status2.initially\_created column:

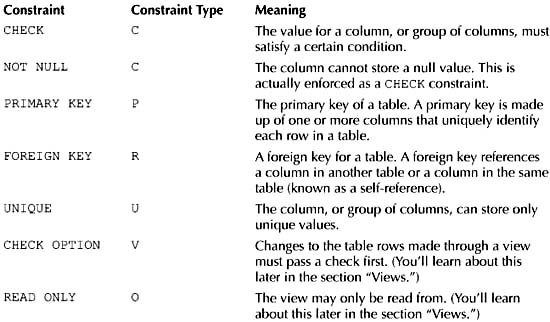
**ALTER TABLE order\_status2**

**DROP COLUMN initially\_created;**

**Adding a Constraint**

In earlier chapters, you’ve seen examples of tables with PRIMARY KEY, FOREIGN KEY, and NOT NULLconstraints. These constraints, along with the other types of constraints, are summarized in [Table 10-3](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch10.html#table_10-3).

You’ll see how to add some of the constraints shown in [Table 10-3](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch10.html#table_10-3) in the following sections.



**TABLE 10-3** *Constraints and Their Meaning*

**Adding a CHECK Constraint**

The following ALTER TABLE statement adds a CHECK constraint to the order\_status2 table:

**ALTER TABLE order\_status2**

**ADD CONSTRAINT order\_status2\_status\_ck**

**CHECK (status IN ('PLACED', 'PENDING', ’SHIPPED'));**

This constraint ensures the status column is always set to PLACED, PENDING, or SHIPPED. The following INSERT adds a row to the order\_status2 table (status is set to PENDING):

**INSERT INTO order\_status2 (**

**id, status, last\_modified, modified\_by**

**) VALUES (**

**1, 'PENDING', '01-JAN-2005', 1**

**);**

If you attempt to add a row that doesn’t satisfy the CHECK constraint, the database returns the error ORA-02290. For example, the following INSERT attempts to add a row whose status is not in the list:

**INSERT INTO order\_status2 (**

**id, status, last\_modified, modified\_by**

**) VALUES (**

**2, 'CLEARED', '01-JAN-2005', 2**

**);**

INSERT INTO order\_status2 (

\*

ERROR at line 1:

ORA-02290: check constraint (STORE.ORDER\_STATUS2\_STATUS\_CK) violated

Because the CHECK constraint is violated, the database rejects the new row.

You can use other comparison operators with a CHECK constraint. The next example adds a CHECK constraint that enforces that the id value is greater than zero:

**ALTER TABLE order\_status2**

**ADD CONSTRAINT order\_status2\_id\_ck CHECK (id > 0);**

When adding a constraint, the existing rows in the table must satisfy the constraint. For example, if the order\_status2 table had rows in it, then the id column for the rows would need to be greater than zero.

Image

**NOTE**  
*There are exceptions to the rule requiring that existing rows satisfy the constraint. You can disable a constraint when you initially add it, and you can set a constraint to apply only to new data, by specifying* ENABLE NOVALIDATE. *You’ll learn more about this later*.

**Adding a NOT NULL Constraint**

The following ALTER TABLE statement adds a NOT NULL constraint to the status column of the order\_status2 table:

**ALTER TABLE order\_status2**

**MODIFY status CONSTRAINT order\_status2\_status\_nn NOT NULL;**

Notice that you use MODIFY to add a NOT NULL constraint rather than ADD CONSTRAINT. The next example adds a NOT NULL constraint to the modified\_by column:

**ALTER TABLE order\_status2**

**MODIFY modified\_by CONSTRAINT order\_status2\_modified\_by\_nn NOT NULL;**

The following example adds a NOT NULL constraint to the last\_modified column:

**ALTER TABLE order\_status2**

**MODIFY last\_modified NOT NULL;**

Notice that I didn’t supply a name for this constraint. In this case, the database automatically assigns an unfriendly name to the constraint, like SYS\_C003381.

Image

**TIP**  
*Always specify a meaningful name to your constraints. That way, when a constraint error occurs, you can easily identify the problem*.

**Adding a FOREIGN KEY Constraint**

Before you see an example of adding a FOREIGN KEY constraint, the following ALTER TABLE statement drops the order\_status2.modified\_by column:

**ALTER TABLE order\_status2**

**DROP COLUMN modified\_by;**

The next statement adds a FOREIGN KEY constraint that references the employees.employee\_id column:

**ALTER TABLE order\_status2**

**ADD CONSTRAINT order\_status2\_modified\_by\_fk**

**modified\_by REFERENCES employees(employee\_id);**

You use the ON DELETE CASCADE clause with a FOREIGN KEY constraint to specify that when a row in the parent table is deleted, any matching rows in the child table are also deleted. The following example drops the modified\_by column and rewrites the previous example to include the ON DELETE CASCADE clause:

**ALTER TABLE order\_status2**

**DROP COLUMN modified\_by;**

**ALTER TABLE order\_status2**

**ADD CONSTRAINT order\_status2\_modified\_by\_fk**

**modified\_by REFERENCES employees(employee\_id) ON DELETE CASCADE;**

When a row is deleted from the employees table, any matching rows in order\_status2 are also deleted.

You use the ON DELETE SET NULL clause with a FOREIGN KEY constraint to specify that when a row in the parent table is deleted, the foreign key column for the row (or rows) in the child table is set to null. The following example drops the modified\_by column from order\_status2 and rewrites the previous example to include the ON DELETE SET NULL clause:

**ALTER TABLE order\_status2**

**DROP COLUMN modified\_by;**

**ALTER TABLE order\_status2**

**ADD CONSTRAINT order\_status2\_modified\_by\_fk**

**modified\_by REFERENCES employees(employee\_id) ON DELETE SET NULL;**

When a row is deleted from the employees table, the modified\_by column for any matching rows in order\_status2 is set to null.

To clean up before moving onto the next section, the following statement drops the modified\_by column:

**ALTER TABLE order\_status2**

**DROP COLUMN modified\_by;**

**Adding a UNIQUE Constraint**

The following ALTER TABLE statement adds a UNIQUE constraint to the order\_status2.statuscolumn:

**ALTER TABLE order\_status2**

**ADD CONSTRAINT order\_status2\_status\_uq UNIQUE (status);**

Any existing or new rows must always have a unique value in the status column.

**Dropping a Constraint**

You drop a constraint using the DROP CONSTRAINT clause of ALTER TABLE. The following example drops the order\_status2\_status\_uq constraint:

**ALTER TABLE order\_status2**

**DROP CONSTRAINT order\_status2\_status\_uq;**

**Disabling a Constraint**

By default, a constraint is enabled when you create it. You can initially disable a constraint by adding DISABLEto the end of the CONSTRAINT clause. The following example adds a constraint to order\_status2, but also disables it:

**ALTER TABLE order\_status2**

**ADD CONSTRAINT order\_status2\_status\_uq UNIQUE (status) DISABLE;**

You can disable an existing constraint using the DISABLE CONSTRAINT clause of ALTER TABLE. The following example disables the order\_status2\_status\_nn constraint:

**ALTER TABLE order\_status2**

**DISABLE CONSTRAINT order\_status2\_status\_nn;**

You can add CASCADE after DISABLE CONSTRAINT to disable all constraints that depend on the specified constraint. You use CASCADE when disabling a primary key or unique constraint that is part of a foreign key constraint of another table.

**Enabling a Constraint**

You can enable an existing constraint using the ENABLE CONSTRAINT clause of ALTER TABLE. The following example enables the order\_status2\_status\_uq constraint:

**ALTER TABLE order\_status2**

**ENABLE CONSTRAINT order\_status2\_status\_uq;**

To enable a constraint, all the rows in the table must satisfy the constraint. For example, if the order\_status2table contained rows, then the status column would have to contain unique values.

You can apply a constraint to new data only by specifying ENABLE NOVALIDATE; for example:

**ALTER TABLE order\_status2**

**ENABLE NOVALIDATE CONSTRAINT order\_status2\_status\_uq;**

Image

**NOTE**  
*The default is* ENABLE VALIDATE, *which means existing rows must pass the constraint check*.

**Deferred Constraints**

A deferred constraint is one that is enforced when a transaction is committed; you use the DEFERRABLE clause when you initially add the constraint. Once you’ve added a constraint, you cannot change it to DEFERRABLE; instead, you must drop and re-create the constraint.

When you add a DEFERRABLE constraint, you can mark it as INITIALLY IMMEDIATE or INITIALLY DEFERRED. Marking as INITIALLY IMMEDIATE means that the constraint is checked whenever you add, update, or delete rows from a table (this is the same as the default behavior of a constraint). INITIALLY DEFERRED means that the constraint is only checked when a transaction is committed. Let’s take a look at an example.

The following statement drops the order\_status2\_status\_uq constraint:

**ALTER TABLE order\_status2**

**DROP CONSTRAINT order\_status2\_status\_uq;**

The next example adds the order\_status2\_status\_uq constraint, setting it to DEFERRABLE INITIALLY DEFERRED:

**ALTER TABLE order\_status2**

**ADD CONSTRAINT order\_status2\_status\_uq UNIQUE (status)**

**DEFERRABLE INITIALLY DEFERRED;**

If you add rows to order\_status2, the order\_status2\_status\_uq constraint isn’t enforced until you perform a commit.

**Getting Information on Constraints**

You can retrieve information on your constraints by querying the user\_constraints view. [Table 10-4](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch10.html#table_10-4)describes some of the columns in user\_constraints.

Image

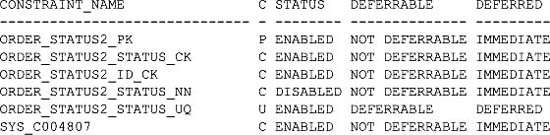
**NOTE**  
*You can retrieve information on all the constraints you have access to by querying the*all\_constraints *view*.

The following example retrieves some of the columns from user\_constraints for the order\_status2table:

**SELECT constraint\_name, constraint\_type, status, deferrable, deferred**

**FROM user\_constraints**

**WHERE table\_name='ORDER\_STATUS2';**



Notice that all the constraints except one have a helpful name. One constraint has the database-generated name of SYS\_C004807 (this name is automatically generated, and it will be different in your database). This constraint is the one for which I omitted the name when creating it earlier.

Image

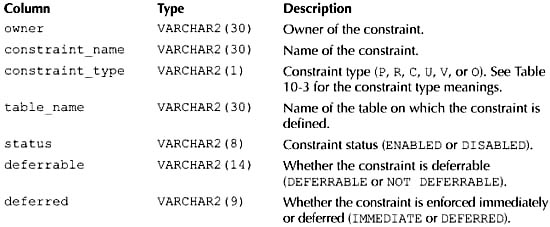
**TIP**  
*Always add a descriptive name for your constraints*.

**Getting Information on the Constraints for a Column**

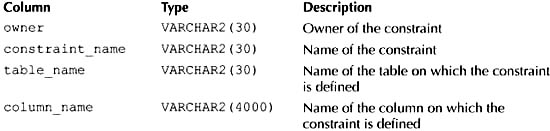
You can retrieve information on the constraints for a column by querying the user\_cons\_columns view. [Table 10-5](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch10.html#table_10-5) describes some of the columns in user\_cons\_columns.

Image

**NOTE**  
*You can retrieve information on all the column constraints you have access to by querying the*all\_cons\_columns *view*.



**TABLE 10-4** *Some Columns in the user\_constraints View*



**TABLE 10-5** *Some Columns in the user\_cons\_columns View*

The following example retrieves the constraint\_name and column\_name from user\_cons\_columnsfor the order\_status2 table:

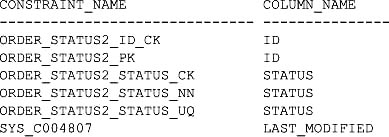
**COLUMN column\_name FORMAT a15**

**SELECT constraint\_name, column\_name**

**FROM user\_cons\_columns**

**WHERE table\_name = 'ORDER\_STATUS2'**

**ORDER BY constraint\_name;**



The next query joins user\_constraints and user\_cons\_columns to get the column\_name, constraint\_name, constraint\_type, and status:

**SELECT ucc.column\_name, ucc.constraint\_name, uc.constraint\_type, uc.status**

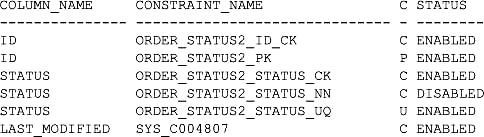
**FROM user\_constraints uc, user\_cons\_columns ucc**

**WHERE uc.table\_name = ucc.table\_name**

**AND uc.constraint\_name = ucc.constraint\_name**

**AND ucc.table\_name = 'ORDER\_STATUS2'**

**ORDER BY ucc.constraint\_name;**



**Renaming a Table**

You rename a table using the RENAME statement. The following example renames order\_status2 to order\_state:

**RENAME order\_status2 TO order\_state;**

Image

**NOTE**  
*If you have used the table name in your constraint names, then you should change the names of your constraints*.

The next example changes the table name back to the original:

**RENAME order\_state TO order\_status2;**

**Adding a Comment to a Table**

A comment can help you remember what the table or column is used for. You add a comment table or column using the COMMENT statement. The following example adds a comment to the order\_status2 table:

**COMMENT ON TABLE order\_status2 IS**

**'order\_status2 stores the state of an order';**

The next example adds a comment to the order\_status2.last\_modified column:

**COMMENT ON COLUMN order\_status2.last\_modified IS**

**'last\_modified stores the date and time the order was modified last';**

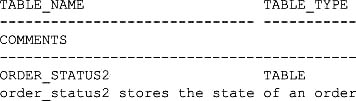
**Retrieving Table Comments**

You can retrieve the comments on your tables from the user\_tab\_comments view, as shown here:

**SELECT \***

**FROM user\_tab\_comments**

**WHERE table\_name = 'ORDER\_STATUS2';**



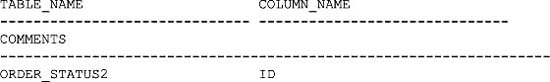
**Retrieving Column Comments**

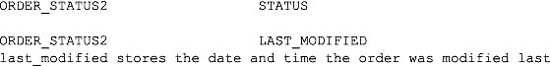
You can retrieve the comments on your columns from the user\_col\_comments view; for example:

**SELECT \***

**FROM user\_col\_comments**

**WHERE table\_name = 'ORDER\_STATUS2';**





**Truncating a Table**

You truncate a table using the TRUNCATE statement. This removes *all* the rows from a table and resets the storage area for a table. The following example truncates order\_status2:

**TRUNCATE TABLE order\_status2;**

Image

**TIP**  
*If you need to remove all the rows from a table, you should use* TRUNCATE *rather than* DELETE. *This is because* TRUNCATE *resets the storage area for a table ready to receive new rows. A* TRUNCATE*statement doesn’t require any undo space in the database, and you don’t have to run a* COMMIT *to make the delete permanent. Undo space is an area that the database software uses to record database changes*.

**Dropping a Table**

You drop a table using the DROP TABLE statement. The following example drops the order\_status2 table:

**DROP TABLE order\_status2;**

This concludes the discussion of tables. In the next section, you’ll learn about sequences.

**SEQUENCES**

A *sequence* is a database item that generates a sequence of integers. You typically use the integers generated by a sequence to populate a numeric primary key column. In this section, you’ll learn how to

Image Create a sequence.

Image Retrieve information on a sequence from the data dictionary.

Image Use a sequence.

Image Modify a sequence.

Image Drop a sequence.

**Creating a Sequence**

You create a sequence using the CREATE SEQUENCE statement, which has the following syntax:

CREATE SEQUENCE *sequence\_name*

[START WITH *start\_num*]

[INCREMENT BY *increment\_num*]

[ { MAXVALUE *maximum\_num* | NOMAXVALUE } ]

[ { MINVALUE *minimum\_num* | NOMINVALUE } ]

[ { CYCLE | NOCYCLE } ]

[ { CACHE *cache\_num* | NOCACHE } ]

[ { ORDER | NOORDER } ];

where

Image *sequence\_name* is the name of the sequence.

Image *start\_num* is the integer to start the sequence. The default start number is 1.

Image *increment\_num* is the integer to increment the sequence by. The default increment number is 1. The absolute value of *increment\_num* must be less than the difference between *maximum\_num* and *minimum\_num*.

Image *maximum\_num* is the maximum integer of the sequence; *maximum\_num* must be greater than or equal to *start\_num*, and *maximum\_num* must be greater than *minimum\_num*.

Image NOMAXVALUE specifies the maximum is 1027 for an ascending sequence or −1 for a descending sequence. NOMAXVALUE is the default.

Image *minimum\_num* is the minimum integer of the sequence; *minimum\_num* must be less than or equal to *start\_num*, and *minimum\_num* must be less than *maximum\_num*.

Image NOMINVALUE specifies the minimum is 1 for an ascending sequence or −1026 for a descending sequence. NOMINVALUE is the default.

Image CYCLE means the sequence generates integers even after reaching its maximum or minimum value. When an ascending sequence reaches its maximum value, the next value generated is the minimum. When a descending sequence reaches its minimum value, the next value generated is the maximum.

Image NOCYCLE means the sequence cannot generate any more integers after reaching its maximum or minimum value. NOCYCLE is the default.

Image *cache\_num* is the number of integers to keep in memory. The default number of integers to cache is 20. The minimum number of integers that may be cached is 2. The maximum integers that may be cached is determined by the formula CEIL (*maximum\_num* - *minimum\_num*)/ABS(*increment\_num*).

Image NOCACHE means no caching. This stops the database from pre-allocating values for the sequence, which prevents numeric gaps in the sequence but reduces performance. Gaps occur because cached values are lost when the database is shut down. If you omit CACHE and NOCACHE, the database caches 20 sequence numbers by default.

Image ORDER guarantees the integers are generated in the order of the request. You typically use ORDER when using Real Application Clusters, which are set up and managed by database administrators. Real Application Clusters are multiple database servers that share the same memory. Real Application Clusters can improve performance.

Image NOORDER doesn’t guarantee the integers are generated in the order of the request. NOORDER is the default.

The following example connects as the store user and creates a sequence named s\_test (I always put s\_ at the beginning of sequences):

**CONNECT store/store\_password**

**CREATE SEQUENCE s\_test;**

Because this CREATE SEQUENCE statement omits the optional parameters, the default values are used. This means that *start\_num* and *increment\_num* are set to the default of 1.

The next example creates a sequence named s\_test2 and specifies values for the optional parameters:

**CREATE SEQUENCE s\_test2**

**START WITH 10 INCREMENT BY 5**

**MINVALUE 10 MAXVALUE 20**

**CYCLE CACHE 2 ORDER;**

The final example creates a sequence named s\_test3 that starts at 10 and counts down to 1:

**CREATE SEQUENCE s\_test3**

**START WITH 10 INCREMENT BY -1**

**MINVALUE 1 MAXVALUE 10**

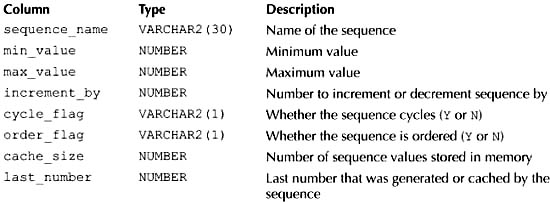
**CYCLE CACHE 5;**

**Retrieving Information on Sequences**

You can retrieve information on your sequences from the user\_sequences view. [Table 10-6](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch10.html#table_10-6) describes the columns in user\_sequences.

Image

**NOTE**  
*You can retrieve information on all the sequences you have access to by querying the*all\_sequences *view*.



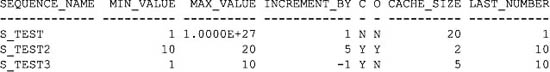
**TABLE 10-6** *Some Columns in the user\_sequences View*

The following example retrieves the details for the sequences from user\_sequences:

**COLUMN sequence\_name FORMAT a13**

**SELECT \* FROM user\_sequences**

**ORDER BY sequence\_name;**



**Using a Sequence**

A sequence generates a series of numbers. A sequence contains two pseudo columns named currval and nextval that you use to get the current value and the next value from the sequence.

Before retrieving the current value, you must first initialize the sequence by retrieving the next value. When you select s\_test.nextval the sequence is initialized to 1. For example, the following query retrieves s\_test.nextval; notice that the dual table is used in the FROM clause:

**SELECT s\_test.nextval**

**FROM dual;**

NEXTVAL

----------

1

The first value in the s\_test sequence is 1. Once the sequence is initialized, you can get the current value from the sequence by retrieving currval. For example:

**SELECT s\_test.currval**

**FROM dual;**

CURRVAL

----------

1

When you retrieve currval, nextval remains unchanged; nextval only changes when you retrieve nextval to get the next value. The following example retrieves s\_test.nextval and s\_test.currval; notice that these values are both 2:

**SELECT s\_test.nextval, s\_test.currval**

**FROM dual;**

NEXTVAL CURRVAL

--------- ----------

2 2

Retrieving s\_test.nextval gets the next value in the sequence, which is 2; s\_test.currval is also 2.

The next example initializes s\_test2 by retrieving s\_test2.nextval; notice that the first value in the sequence is 10:

**SELECT s\_test2.nextval**

**FROM dual;**

NEXTVAL

----------

10

The maximum value for s\_test2 is 20, and the sequence was created with the CYCLE option, meaning that the sequence will cycle back to 10 once it reaches the maximum of 20:

**SELECT s\_test2.nextval**

**FROM dual;**

NEXTVAL

----------

15

**SELECT s\_test2.nextval**

**FROM dual;**

NEXTVAL

----------

20

**SELECT s\_test2.nextval**

**FROM dual;**

NEXTVAL

----------

10

The s\_test3 sequence starts at 10 and counts down to 1:

**SELECT s\_test3.nextval**

**FROM dual;**

NEXTVAL

----------

10

**SELECT s\_test3.nextval**

**FROM dual;**

NEXTVAL

----------

9

**SELECT s\_test3.nextval**

**FROM dual;**

NEXTVAL

----------

8

**Populating a Primary Key Using a Sequence**

Sequences are useful for populating integer primary key column values. Let’s take a look at an example. The following statement re-creates the order\_status2 table:

**CREATE TABLE order\_status2 (**

**id INTEGER CONSTRAINT order\_status2\_pk PRIMARY KEY,**

**status VARCHAR2(10),**

**last\_modified DATE DEFAULT SYSDATE**

**);**

Next, the following statement creates a sequence named s\_order\_status2 (this sequence will be used to populate the order\_status2.id column shortly):

**CREATE SEQUENCE s\_order\_status2 NOCACHE;**

Image

**TIP**  
*When using a sequence to populate a primary key column, you should typically use* NOCACHE *to avoid gaps in the sequence of numbers (gaps occur because cached values are lost when the database is shut down). However, using* NOCACHE *reduces performance. If you are* absolutely sure *you can live with gaps in the primary key values, then consider using* CACHE.

The following INSERT statements add rows to order\_status2; notice that the value for the id column is set using s\_order\_status2.nextval (returns 1 for the first INSERT and 2 for the second INSERT):

**INSERT INTO order\_status2 (**

**id, status, last\_modified**

**) VALUES (**

**s\_order\_status2.nextval, 'PLACED', '01-JAN-2006'**

**);**

**INSERT INTO order\_status2 (**

**id, status, last\_modified**

**) VALUES (**

**s\_order\_status2.nextval, 'PENDING', '01-FEB-2006'**

**);**

The following query retrieves the rows from order\_status2; notice that the id column is set to the first two values (1 and 2) from the s\_order\_status2 sequence:

**SELECT \***

**FROM order\_status2;**

ID STATUS LAST\_MODI

---------- ---------- ---------

1 PLACED 01-JAN-06

2 PENDING 01-FEB-06

**Modifying a Sequence**

You modify a sequence using the ALTER SEQUENCE statement. There are some limitations on what you can modify in a sequence:

Image You cannot change the start value of a sequence.

Image The minimum value cannot be more than the current value of the sequence.

Image The maximum value cannot be less than the current value of the sequence.

The following example modifies s\_test to increment the sequence of numbers by 2:

**ALTER SEQUENCE s\_test**

**INCREMENT BY 2;**

When this is done, the new values generated by s\_test will be incremented by 2. For example, if s\_test.currval is 2, then s\_test.nextval is 4. This is shown in the following example:

**SELECT s\_test.currval**

**FROM dual;**

CURRVAL

----------

2

**SELECT s\_test.nextval**

**FROM dual;**

NEXTVAL

----------

4

**Dropping a Sequence**

You drop a sequence using DROP SEQUENCE. The following example drops s\_test3:

**DROP SEQUENCE s\_test3;**

This concludes the discussion of sequences. In the next section, you’ll learn about indexes.

**INDEXES**

When looking for a particular topic in a book, you can either scan the whole book, or you can use the index to find the location. An index for a database table is similar in concept to a book index, except that database indexes are used to find specific rows in a table. The downside of indexes is that when a row is added to the table, additional time is required to update the index for the new row.

Generally, you should create an index on a column when you are retrieving a small number of rows from a table containing many rows. A good rule of thumb is

*Create an index when a query retrieves <= 10 percent of the total rows in a table*.

This means the column for the index should contain a wide range of values. These types of indexes are called "B-tree" indexes, a name which comes from a tree data structure used in computer science. A good candidate for B-tree indexing would be a column containing a unique value for each row (for example, a social security number). A poor candidate for B-tree indexing would be a column that contains only a small range of values (for example, N, S, E, W or 1, 2, 3, 4, 5, 6). An Oracle database automatically creates a B-tree index for the primary key of a table and for columns included in a unique constraint. For columns that contain a small range of values, you can use a "bitmap" index.

In this section, you’ll learn how to

Image Create a B-tree index.

Image Create a function-based index.

Image Retrieve information on an index from the data dictionary.

Image Modify an index.

Image Drop an index.

Image Create a bitmap index.

**Creating a B-tree Index**

You create a B-tree index using CREATE INDEX, which has the following simplified syntax:

CREATE [UNIQUE] INDEX *index\_name* ON

*table\_name (column\_name*[, *column\_name* …])

TABLESPACE *tab\_space*;

where

Image UNIQUE means that the values in the indexed columns must be unique.

Image *index\_name* is the name of the index.

Image *table\_name* is a database table.

Image *column\_name* is the indexed column. You can create an index on multiple columns (such an index is known as a *composite index*).

Image *tab\_space* is the tablespace for the index. If you don’t provide a tablespace, the index is stored in the user’s default tablespace.

Image

**TIP**  
*For performance reasons, you should typically store indexes in a different tablespace from tables. For simplicity, the examples in this chapter use the default tablespace. In a production database, the database administrator should create separate tablespaces for the tables and indexes*.

I’ll now guide you through the thought processes you should follow when creating a B-tree index for the customers.last\_name column. Assume that the customers table contains a large number of rows and that you regularly retrieve rows using the following type of query:

**SELECT customer\_id, first\_name, last\_name**

**FROM customers**

**WHERE last\_name = 'Brown';**

Also assume that the last\_name column contains somewhat unique values, so that any query using the last\_name column in a WHERE clause will return less than 10 percent of the total number of rows in the table. This means the last\_name column is therefore a good candidate for indexing.

The following CREATE INDEX statement creates an index named i\_customers\_last\_name on the last\_name column of the customers table (I always put i\_ at the start of index names):

**CREATE INDEX i\_customers\_last\_name ON customers(last\_name);**

Once the index has been created, the previous query will take less time to complete.

You can enforce uniqueness of column values using a unique index. For example, the following statement creates a unique index named i\_customers\_phone on the customers.phone column:

**CREATE UNIQUE INDEX i\_customers\_phone ON customers(phone);**

You can also create a composite index on multiple columns. For example, the following statement creates a composite index named i\_employees\_first\_last\_name on the first\_name and last\_namecolumns of the employees table:

**CREATE INDEX i\_employees\_first\_last\_name ON**

**employees(first\_name, last\_name);**

**Creating a Function-Based Index**

In the previous section you saw the index i\_customers\_last\_name. Let’s say you run the following query:

**SELECT first\_name, last\_name**

**FROM customers**

**WHERE last\_name = UPPER('BROWN');**

Because this query uses a function—UPPER(), in this case—the i\_customers\_last\_name index isn’t used. If you want an index to be based on the results of a function, you must create a function-based index, such as:

**CREATE INDEX i\_func\_customers\_last\_name**

**ON customers(UPPER(last\_name));**

In addition, the database administrator must set the initialization parameter QUERY\_REWRITE\_ENABLED to true (the default is false) in order to take advantage of function-based indexes. The following example sets QUERY\_REWRITE\_ENABLED to true:

**CONNECT system/manager**

**ALTER SYSTEM SET QUERY\_REWRITE\_ENABLED = TRUE;**

**Retrieving Information on Indexes**

You can retrieve information on your indexes from the user\_indexes view. [Table 10-7](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch10.html#table_10-7) describes some of the columns in user\_indexes.

Image

**NOTE**  
*You can retrieve information on all the indexes you have access to by querying the* all\_indexes*view*.

The following example connects as the store user and retrieves some of the columns from user\_indexesfor the customers and employees tables; notice that the list of indexes includes customers\_pk, which is a unique index automatically created by the database for the customer\_id primary key column of the customers table:

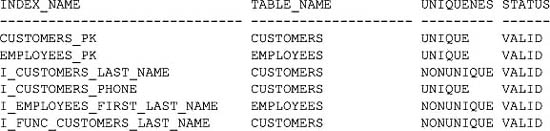
**CONNECT store/store\_password**

**SELECT index\_name, table\_name, uniqueness, status**

**FROM user\_indexes**

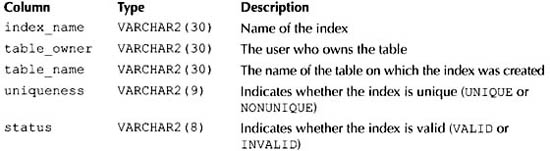
**WHERE table\_name IN ('CUSTOMERS', 'EMPLOYEES')**

**ORDER BY index\_name;**

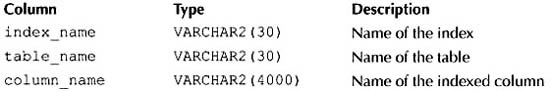


**Retrieving Information on the Indexes on a Column**

You can retrieve information on the indexes on a column by querying the user\_ind\_columns view. [Table 10-8](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch10.html#table_10-8) describes some of the columns in user\_ind\_columns.



**TABLE 10-7** *Some Columns in the user\_indexes View*



**TABLE 10-8** *Some Columns in the user\_ind\_columns View*

Image

**NOTE**  
*You can retrieve information on all the indexes you have access to by querying the*all\_ind\_columns *view*.

The following query retrieves some of the columns from user\_ind\_columns for the customers and employees tables:

**COLUMN table\_name FORMAT a15**

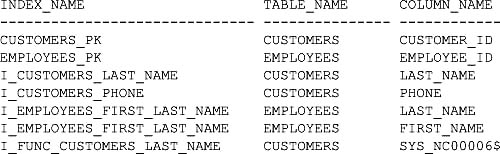
**COLUMN column\_name FORMAT a15**

**SELECT index\_name, table\_name, column\_name**

**FROM user\_ind\_columns**

**WHERE table\_name IN ('CUSTOMERS', 'EMPLOYEES')**

**ORDER BY index\_name;**



**Modifying an Index**

You modify an index using ALTER INDEX. The following example renames the i\_customers\_phone index to i\_customers\_phone\_number:

**ALTER INDEX i\_customers\_phone RENAME TO i\_customers\_phone\_number;**

**Dropping an Index**

You drop an index using the DROP INDEX statement. The following example drops the i\_customers\_phone\_number index:

**DROP INDEX i\_customers\_phone\_number;**

**Creating a Bitmap Index**

Bitmap indexes are typically used in *data warehouses*, which are databases containing very large amounts of data. The data in a data warehouse is typically read using many queries, but the data is not modified by many concurrent transactions. Data warehouses are typically used by organizations for business intelligence analysis, like monitoring sales trends.

A candidate for a bitmap index is a column that is referenced in many queries, but that contains only a small range of values; for example:

Image N, S, E, W

Image 1, 2, 3, 4, 5, 6

Image "Order placed", "Order shipped"

An index basically contains a pointer to a row in a table that contains a given index key value; the key value is used to get the rowid for the row in the table. (As discussed in [Chapter 2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch02.html#ch02), a rowid is used internally by the database to store the physical location of the row.) In a B-tree index, a list of rowids is stored for each key corresponding to the rows with that key value. In a B-tree index, the database stores a list of key values with each rowid, which enables the database to locate an actual row in a table.

In a bitmap index, however, a bitmap is used for each key value; the bitmap enables the database to locate a row. Each bit in the bitmap corresponds to a possible rowid. If the bit is set, then it means that the row with the corresponding rowid contains the key value. A mapping function converts the bit position to an actual rowid.

Bitmap indexes are typically used in tables containing large amounts of data and whose contents are not modified very often. Also, a bitmap index should only be created on columns that contain a small number of distinct values. If the number of distinct values of a column is less than 1 percent of the number of rows in the table, or if the values in a column are repeated more than 100 times, then the column is a candidate for a bitmap index. For example, if you had a table with 1 million rows, a column with 10,000 distinct values or less is a good candidate for a bitmap index; also, updates to the rows in the table should be rare, and the column would need to be frequently used in the WHERE clause of queries.

The following statement creates a bitmap index on the status column of the order\_status table:

CREATE BITMAP INDEX i\_order\_status ON order\_status(status);

Image

**NOTE**  
*Of course, this example is not a real-world example because the* order\_status *table does not contain enough rows*.

You can find more information on bitmap indexes in *Oracle Database Performance Tuning Guide* and *Oracle Database Concepts*, both books published by Oracle Corporation. These books also contain information about other exotic types of indexes you can use.

This concludes the discussion of indexes. In the next section, you’ll learn about views.

**VIEWS**

A view is a predefined query on one or more tables (known as *base tables*). Retrieving information from a view is done in the same manner as retrieving from a table: you simply include the view in the FROM clause of a query. With some views you can also perform DML operations on the base tables.

Image

**NOTE**  
*Views don’t store rows. Rows are always stored in tables*.

You’ve already seen some examples of retrieving information from views when you selected rows from the data dictionary, which is accessed through views—for example, user\_tables, user\_sequences, and user\_indexes are all views.

Views offer several benefits, such as the following:

Image You can put a complex query into a view and grant users access to the view. This allows you to hide complexity from users.

Image You can stop users from directly querying the base tables by granting them access only to the view.

Image You can allow a view to access only certain rows in the base tables. This allows you to hide rows from an end user.

In this section, you’ll learn how to

Image Create and use a view.

Image Get the details of a view from the data dictionary.

Image Modify a view.

Image Drop a view.

**Creating and Using a View**

You create a view using CREATE VIEW, which has the following simplified syntax:

CREATE [OR REPLACE] VIEW [{FORCE | NOFORCE}] VIEW view\_name

[(*alias\_name*[, *alias\_name* …])] AS *subquery*

[WITH {CHECK OPTION | READ ONLY} CONSTRAINT *constraint\_name*];

where

Image OR REPLACE means the view replaces an existing view.

Image FORCE means the view is to be created even if the base tables don’t exist.

Image NOFORCE means the view is not created if the base tables don’t exist. NOFORCE is the default.

Image *view\_name* is the name of the view.

Image *alias\_name* is the name of an alias for an expression in the subquery. There must be the same number of aliases as there are expressions in the subquery.

Image *subquery* is the subquery that retrieves from the base tables. If you’ve supplied aliases, you can use those aliases in the list after the SELECT.

Image WITH CHECK OPTION means that only the rows that would be retrieved by the subquery can be inserted, updated, or deleted. By default, the rows are not checked.

Image *constraint\_name* is the name of the WITH CHECK OPTION or WITH READ ONLY constraint.

Image WITH READ ONLY means the rows may only read from the base tables.

There are two basic types of views:

Image Simple views, which contain a subquery that retrieves from one base table

Image Complex views, which contain a subquery that

Image Retrieves from multiple base tables

Image Groups rows using a GROUP BY or DISTINCT clause

Image Contains a function call

You’ll learn how to create and use these types of views in the following sections.

**Privilege for Views**

In order to create a view, the user must have the CREATE VIEW privilege. The following example connects as the system user and grants the CREATE VIEW privilege to the store user:

**CONNECT system/manager**

**GRANT CREATE VIEW TO store;**

**Creating and Using Simple Views**

Simple views access one base table. The following example connects as the store user and creates a view named cheap\_products\_view whose subquery retrieves products only where the price is less than $15:

**CONNECT store/store\_password**

**CREATE VIEW cheap\_products\_view AS**

**SELECT \***

**FROM products**

**WHERE price < 15;**

The next example creates a view named employees\_view whose subquery retrieves all the columns from the employees table except salary:

**CREATE VIEW employees\_view AS**

**SELECT employee\_id, manager\_id, first\_name, last\_name, title**

**FROM employees;**

**Performing a Query on a View**

Once you’ve created a view, you can use it to access the base table. The following query retrieves rows from cheap\_products\_view:

**SELECT product\_id, name, price**

**FROM cheap\_products\_view;**

PRODUCT\_ID NAME PRICE

---------- ------------------------------ ----------

4 Tank War 13.95

6 2412: The Return 14.95

7 Space Force 9 13.49

8 From Another Planet 12.99

9 Classical Music 10.99

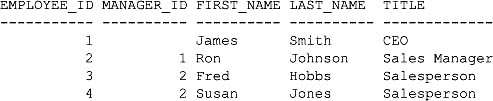
11 Creative Yell 14.99

12 My Front Line 13.49

The next example retrieves rows from employees\_view:

**SELECT \***

**FROM employees\_view;**



**Performing an INSERT Using a View**

You can perform DML statements using cheap\_products\_view. The following example performs an INSERT using cheap\_products\_view and then retrieves the row:

**INSERT INTO cheap\_products\_view (**

**product\_id, product\_type\_id, name, price**

**) VALUES (**

**13, 1, 'Western Front', 13.50**

**);**

1 row created.

**SELECT product\_id, name, price**

**FROM cheap\_products\_view**

**WHERE product\_id = 13;**

PRODUCT\_ID NAME PRICE

---------- ------------------------------ ----------

13 Western Front 13.5

Image

**NOTE**  
*You can perform DML statements only with simple views. Complex views don’t support DML*.

Because cheap\_products\_view didn’t use WITH CHECK OPTION, you can insert, update, and delete rows that aren’t retrievable by the view. The following example inserts a row whose price is $16.50 (this is greater than $15 and therefore not retrievable by the view):

**INSERT INTO cheap\_products\_view (**

**product\_id, product\_type\_id, name, price**

**) VALUES (**

**14, 1, 'Eastern Front', 16.50**

**);**

1 row created.

**SELECT \***

**FROM cheap\_products\_view**

**WHERE product\_id = 14;**

no rows selected

The employees\_view contains a subquery that selects every column from employees except salary. When you perform an INSERT using employees\_view, the salary column in the employees base table will be set to null; for example:

**INSERT INTO employees\_view (**

**employee\_id, manager\_id, first\_name, last\_name, title**

**) VALUES (**

**5, 1, 'Jeff', 'Jones', 'CTO'**

**);**

1 row created.

**SELECT employee\_id, first\_name, last\_name, salary**

**FROM employees**

**WHERE employee\_id = 5;**

EMPLOYEE\_ID FIRST\_NAME LAST\_NAME SALARY

----------- ---------- ---------- ---------

5 Jeff Jones

The salary column is null.

**Creating a View with a CHECK OPTION Constraint**

You can specify that DML statements on a view must satisfy the subquery using a CHECK OPTION constraint. For example, the following statement creates a view named cheap\_products\_view2 that has a CHECK OPTION constraint:

**CREATE VIEW cheap\_products\_view2 AS**

**SELECT \***

**FROM products**

**WHERE price < 15**

**WITH CHECK OPTION CONSTRAINT cheap\_products\_view2\_price;**

The next example attempts to insert a row using cheap\_products\_view2 with a price of $19.50; notice that the database returns an error because the row isn’t retrievable by the view:

**INSERT INTO cheap\_products\_view2 (**

**product\_id, product\_type\_id, name, price**

**) VALUES (**

**15, 1, ’Southern Front', 19.50**

**);**

INSERT INTO cheap\_products\_view2 (

\*

ERROR at line 1:

ORA-01402: view WITH CHECK OPTION where-clause violation

**Creating a View with a READ ONLY Constraint**

You can make a view read-only by adding a READ ONLY constraint to the view. For example, the following statement creates a view named cheap\_products\_view3 that has a READ ONLY constraint:

**CREATE VIEW cheap\_products\_view3 AS**

**SELECT \***

**FROM products**

**WHERE price < 15**

**WITH READ ONLY CONSTRAINT cheap\_products\_view3\_read\_only;**

The following example attempts to insert a row using cheap\_products\_view3; notice that the database returns an error because the view is read-only and doesn’t allow DML statements:

**INSERT INTO cheap\_products\_view3 (**

**product\_id, product\_type\_id, name, price**

**) VALUES (**

**16, 1, 'Northern Front', 19.50**

**);**

product\_id, product\_type\_id, name, price

\*

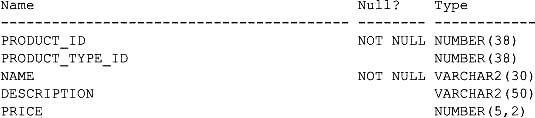
ERROR at line 2:

ORA-42399: cannot perform a DML operation on a read-only view

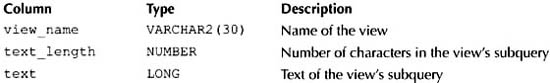
**Getting Information on View Definitions**

You can retrieve information on view definitions using the DESCRIBE command. The following example uses DESCRIBE with cheap\_products\_view3:

**DESCRIBE cheap\_products\_view3**



You can also retrieve information about your views from the user\_views view. [Table 10-9](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch10.html#table_10-9) describes some of the columns in user\_views.



**TABLE 10-9** *Some Columns in the user\_views View*

Image

**NOTE**  
*You can retrieve information on all the indexes you have access to by querying* all\_views.

To see the entire view definition stored in the text column, you use the SQL\*Plus command SET LONG, which sets the number of characters displayed by SQL\*Plus when retrieving LONG columns. For example, the following command sets LONG to 200:

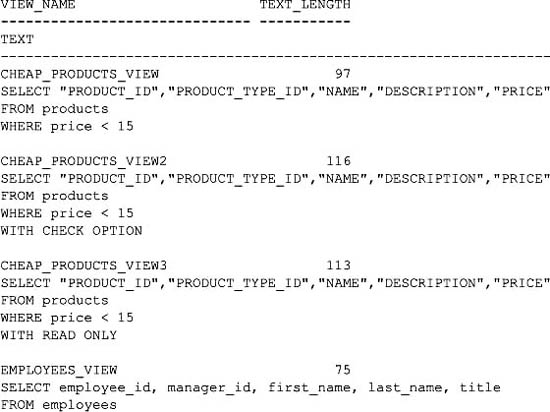
SET LONG 200

The following query retrieves the view\_name, text\_length, and text columns from user\_views:

**SELECT view\_name, text\_length, text**

**FROM user\_views**

**ORDER BY view\_name;**



**Retrieving Information on View Constraints**

Earlier you saw that you can add CHECK OPTION and READ ONLY constraints to a view; cheap\_products\_view2 contained a CHECK OPTION constraint to ensure the price was less than $15; cheap\_products\_view3 contained a READ ONLY constraint to prevent modifications to the rows in the base table.

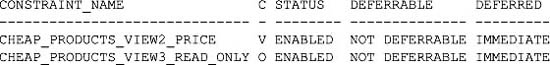
You retrieve information on view constraints from the user\_constraints view; for example:

**SELECT constraint\_name, constraint\_type, status, deferrable, deferred**

**FROM user\_constraints**

**WHERE table\_name IN ('CHEAP\_PRODUCTS\_VIEW2', 'CHEAP\_PRODUCTS\_VIEW3')**

**ORDER BY constraint\_name;**



The constraint\_type for CHEAP\_PRODUCTS\_VIEW2\_PRICE is V, which, as shown earlier in [Table 10-3](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch10.html#table_10-3), corresponds to a CHECK OPTION constraint. The constraint\_type for CHEAP\_PRODUCTS\_VIEW3\_READ\_ONLY is O, which corresponds to a READ ONLY constraint.

**Creating and Using Complex Views**

Complex views contain subqueries that

Image Retrieve rows from multiple base tables.

Image Group rows using a GROUP BY or DISTINCT clause.

Image Contain a function call.

The following example creates a view named products\_and\_types\_view whose subquery performs a full outer join on the products and product\_types tables using the SQL/92 syntax:

**CREATE VIEW products\_and\_types\_view AS**

**SELECT p.product\_id, p.name product\_name, pt.name product\_type\_name, p.price**

**FROM products p FULL OUTER JOIN product\_types pt**

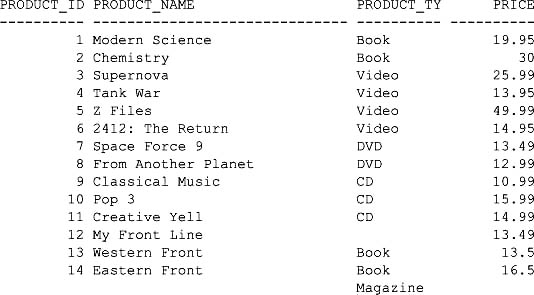
**USING (product\_type\_id)**

**ORDER BY p.product\_id;**

The following example queries products\_and\_types\_view:

**SELECT \***

**FROM products\_and\_types\_view;**



The next example creates a view named employee\_salary\_grades\_view whose subquery uses an inner join to retrieve the salary grades for the employees:

**CREATE VIEW employee\_salary\_grades\_view AS**

**SELECT e.first\_name, e.last\_name, e.title, e.salary, sg.salary\_grade\_id**

**FROM employees e INNER JOIN salary\_grades sg**

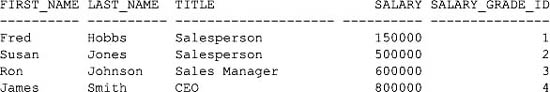
**ON e.salary BETWEEN sg.low\_salary AND sg.high\_salary**

**ORDER BY sg.salary\_grade\_id;**

The following example queries employee\_salary\_grades\_view:

**SELECT \***

**FROM employee\_salary\_grades\_view;\B**



The next example creates a view named product\_average\_view whose subquery uses

Image A WHERE clause to filter the rows from the products table to those whose price is less than $15.

Image A GROUP BY clause to group the remaining rows by the product\_type\_id column.

Image A HAVING clause to filter the row groups to those whose average price is greater than $13.

**CREATE VIEW product\_average\_view AS**

**SELECT product\_type\_id, AVG(price) average\_price**

**FROM products**

**WHERE price < 15**

**GROUP BY product\_type\_id**

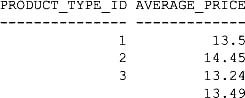
**HAVING AVG(price) > 13**

**ORDER BY product\_type\_id;**

The following example queries product\_average\_view:

**SELECT \***

**FROM product\_average\_view;**



**Modifying a View**

You can completely replace a view using CREATE OR REPLACE VIEW. The following example uses CREATE OR REPLACE VIEW to replace product\_average\_view:

**CREATE OR REPLACE VIEW product\_average\_view AS**

**SELECT product\_type\_id, AVG(price) average\_price**

**FROM products**

**WHERE price < 12**

**GROUP BY product\_type\_id**

**HAVING AVG(price) > 11**

**ORDER BY product\_type\_id;**

You can alter the constraints on a view using ALTER VIEW. The following example uses ALTER VIEW to drop the cheap\_products\_view2\_price constraint from cheap\_products\_view2:

**ALTER VIEW cheap\_products\_view2**

**DROP CONSTRAINT cheap\_products\_view2\_price;**

**Dropping a View**

You drop a view using DROP VIEW. The following example drops cheap\_products\_view2:

**DROP VIEW cheap\_products\_view2;**

This concludes the discussion of views. In the next section, you’ll learn about flashback data archives.

**FLASHBACK DATA ARCHIVES**

Flashback data archives, which are new for Oracle Database 11*g*, store changes made to a table over a period of time and provide you with a full audit trail. Once you’ve created a flashback archive and added a table to it you can do the following:

Image View rows as they were at a specific timestamp

Image View rows as they were between two timestamps

You create a flashback archive using the CREATE FLASHBACK ARCHIVE statement. The following example connects as the system user and creates a flashback archive named test\_archive:

**CONNECT system/manager**

**CREATE FLASHBACK ARCHIVE test\_archive**

**TABLESPACE example**

**QUOTA 1 M**

**RETENTION 1 DAY;**

Notice the following:

Image The archive is created in the example tablespace; you can see the full list of tablespaces by running the query SELECT tablespace\_name FROM dba\_tablespaces.

Image The test\_archive has a quota of 1 megabyte, which means it can store up to 1 megabyte of data in the example tablespace.

Image The data in test\_archive is retained for 1 day, after which time the data is purged.

You may alter an existing table to store data in the archive; for example:

**ALTER TABLE store.products FLASHBACK ARCHIVE test\_archive;**

Any subsequent changes made to the store.products table are now recorded in the archive. The following INSERT statement adds a row to the store.products table:

**INSERT INTO store.products (**

**product\_id, product\_type\_id, name, description, price**

**) VALUES (**

**15, 1, 'Using Linux', 'How to Use Linux', 39.99**

**);**

The following query retrieves this row:

**SELECT product\_id, name, price**

**FROM store.products**

**WHERE product\_id = 15;**

PRODUCT\_ID NAME PRICE

---------- ------------------------------ ----------

15 Using Linux 39.99

You can view the rows as they were 5 minutes ago using the following query:

**SELECT product\_id, name, price**

**FROM store.products**

**AS OF TIMESTAMP**

**(SYSTIMESTAMP - INTERVAL '5' MINUTE);**



Notice that the new row is missing. This is because it was added to the table after the date and time specified in the query (assuming the previous INSERT was run less than 5 minutes ago).

You can also view the rows as they were at a specific timestamp using the following query (if you run this query, you need to change the timestamp to a date and time before you ran the INSERT statement earlier):

**SELECT product\_id, name, price**

**FROM store.products**

**AS OF TIMESTAMP**

**TO\_TIMESTAMP('2007-08-12 13:05:00', 'YYYY-MM-DD HH24:MI:SS');**

The new row will be missing from the results again, because it was added to the table after the date and time specified in the query.

You can view the rows as they were between two timestamps using the following query (you need to change the timestamps):

**SELECT product\_id, name, price**

**FROM store.products VERSIONS BETWEEN TIMESTAMP**

**TO\_TIMESTAMP('2007-08-12 12:00:00', 'YYYY-MM-DD HH24:MI:SS')**

**AND TO\_TIMESTAMP('2007-08-12 12:59:59', 'YYYY-MM-DD HH24:MI:SS');**

You can view the rows as they were between one timestamp and the present time using the following query (you need to change the timestamp):

**SELECT product\_id, name, price**

**FROM store.products VERSIONS BETWEEN TIMESTAMP**

**TO\_TIMESTAMP('2007-08-12 13:45:52', 'YYYY-MM-DD HH24:MI:SS')**

**AND MAXVALUE;**

You can stop archiving of data for a table using ALTER TABLE; for example:

**ALTER TABLE store.products NO FLASHBACK ARCHIVE;**

When you create a table, you can specify a flashback archive for that table; for example:

**CREATE TABLE store.test\_table (**

**id INTEGER,**

**name VARCHAR2(10)**

**) FLASHBACK ARCHIVE test\_archive;**

You can view the details for an archive using the following views:

Image user\_flashback\_archive and dba\_flashback\_archive, which display general information about the flashback archives

Image user\_flashback\_archive\_ts and dba\_flashback\_archive\_ts, which display information about the tablespaces containing the flashback archives

Image user\_flashback\_archive\_tables and dba\_flashback\_archive\_tables, which display information about the tables that are enabled for flashback archiving

You can alter a flashback archive; for example, the following statement changes the data retention period to 2 years:

**ALTER FLASHBACK ARCHIVE test\_archive**

**MODIFY RETENTION 2 YEAR;**

You can purge the data from a flashback archive before a given timestamp; for example, the following statement purges data older than 1 day:

**ALTER FLASHBACK ARCHIVE test\_archive**

**PURGE BEFORE TIMESTAMP(SYSTIMESTAMP – INTERVAL '1' DAY);**

You can purge all the data in a flashback archive; for example:

**ALTER FLASHBACK ARCHIVE test\_archive PURGE ALL;**

You can drop a flashback archive; for example:

**DROP FLASHBACK ARCHIVE test\_archive;**

Image

**NOTE**  
*Go ahead and rerun* store\_schema.sql *to re-create the store tables so that your queries match mine in the rest of this book*.

**SUMMARY**

In this chapter, you have learned the following:

Image A table is created using the CREATE TABLE statement.

Image A sequence generates a sequence of integers.

Image A database index can speed up access to rows.

Image A view is a predefined query on one or more base tables.

Image A flashback data archive stores changes made to a table over a period of time.