**CHAPTER  
14  
  
SOME COMPLEX POSSIBILITIES**

**T**his chapter continues the study of the more complex Oracle functions and features. Of particular interest here is the creation of simple and group queries that can be turned into views, the use of totals in calculations, and the creation of reports showing a tree structure from one or more tables. Like the techniques covered in [Chapter 13](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch13.html#ch13), these techniques are not essential for most reporting needs. If they look overly difficult, don’t be frightened off. If you are new to Oracle and the use of its query capabilities, it is enough to know that these capabilities exist and you can turn to them if needed.

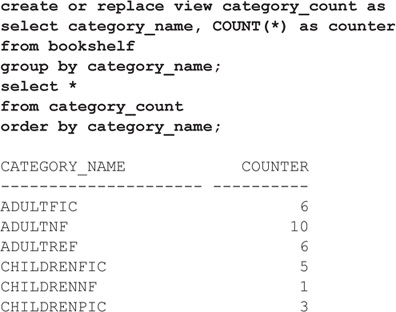
**Complex Groupings**

Views can build on each other. In [Chapter 12](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch12.html#ch12), you saw the concept of creating a view of a grouping of rows from a table. As shown in [Chapter 12](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch12.html#ch12), you can easily join views to other views and tables to produce additional views to simplify the tasks of querying and reporting.

As your groupings grow more complex, you will find that views are invaluable to your coding efforts; they simplify the representation of data at different grouping levels within your application. They also make it easier to use the more advanced analytic functions available.

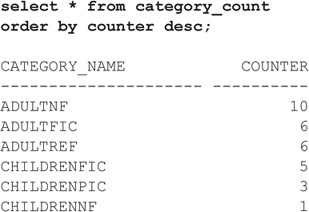
Consider the CATEGORY\_COUNT view, first encountered in [Chapter 12](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch12.html#ch12):

Image



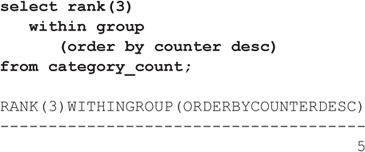
Let’s order the results by their COUNTER column values, with the highest first:

Image



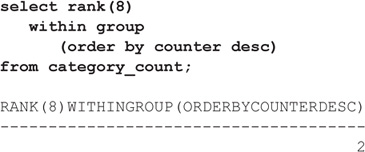
The output shows the ranking of the categories; the ADULTNF category ranks first in terms of the number of books. Without displaying this list, you could determine where a different COUNTER value would be in the rankings. To do this, we’ll use the RANK built-in function. As shown in the following listing, the RANK function takes a value as its input and has additional clauses—the WITHIN GROUP and ORDER BY clauses—that tell Oracle how to do the ranking. Where would a COUNTER value of 3 rank?

Image



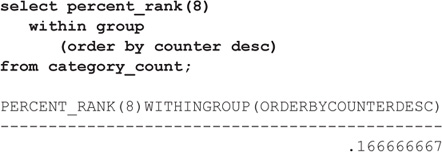
A COUNTER value of 3 would be the fifth-highest COUNTER value. How about a COUNTER value of 8?

Image



Adding those five books to the category would move it up to second place. From a percentile perspective, what would the ranking be for that category?

Image



As expected, it would be in the top one-sixth of the categories.

With this technique of using both summary views and analytic functions, you can create views and reports that include weighted average, effective yield, percentage of total, percentage of subtotal, and many similar calculations. There is no effective limit to how many views can be built on top of each other, although even the most complex calculations seldom require more than three or four levels of views built on views. Note that you can also create inline views in the FROM clause, as shown in [Chapter 12](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch12.html#ch12).

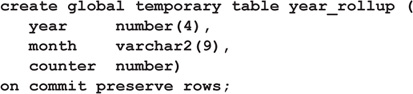
**Using Temporary Tables**

You can create a table that exists solely for your session or whose data persists for the duration of your transaction. You can use temporary tables to support specialized rollups or specific application-processing requirements whose results will not persist beyond the session or even past a COMMIT statement.

To create a temporary table, use the CREATE GLOBAL TEMPORARY TABLE command. When you create a temporary table, you can specify whether it should last for the duration of your session (via the ON COMMIT PRESERVE ROWS clause) or whether its rows should be deleted when the transaction completes (via the ON COMMIT DELETE ROWS clause).

Unlike a permanent table, a temporary table does not automatically allocate space when it is created. Space will be dynamically allocated for the table as rows are inserted:

Image



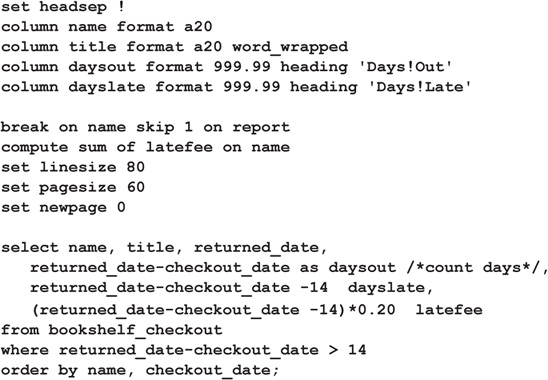
You can see the duration of your data in YEAR\_ROLLUP by querying the DURATION column of USER\_TABLES for this table. In this case, the value of DURATION is SYS$SESSION. If ON COMMIT DELETE ROWS had been specified instead, the DURATION value would be SYS$TRANSACTION.

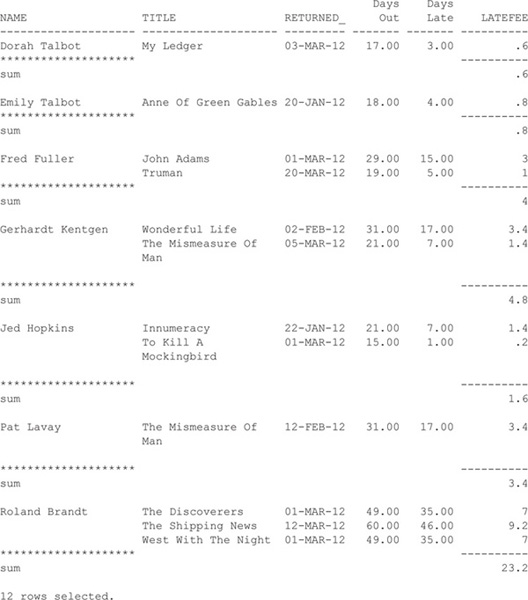
Now that the YEAR\_ROLLUP table exists, you can populate it, for instance, via an INSERT AS SELECT command with a complex query. You can then query the YEAR\_ROLLUP table as part of a join with other tables. You may find this method simpler to implement than the methods shown in [Chapter 12](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch12.html#ch12)for creating views.

**Using ROLLUP, GROUPING, and CUBE**

How can you perform grouping operations, such as totals, within a single SQL statement rather than via SQL\*Plus commands? You can use the ROLLUP and CUBE functions to enhance the grouping actions performed within your queries. Let’s see how this enables us to manage the data related to book returns. The book loaner program has become more popular, so the loan time is now limited to 14 days, with a $0.20 fee per extra day. The following report shows the late charges by person:

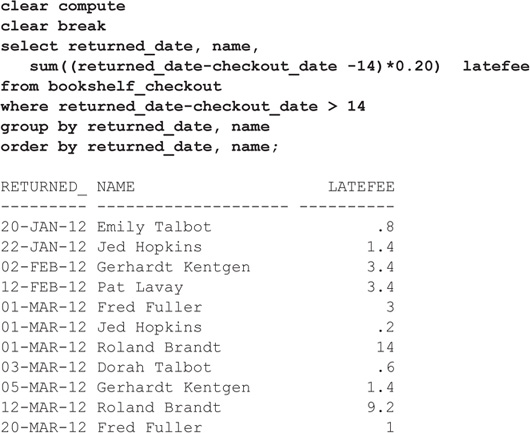
Image





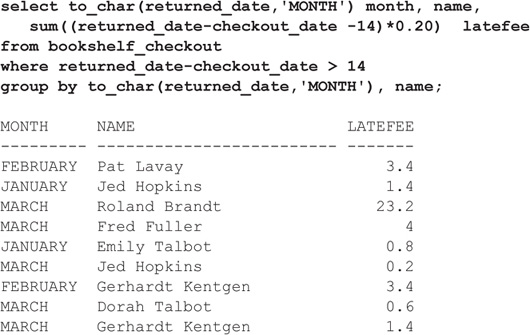
We can eliminate the DAYSOUT derived column display and focus on the late fees, showing the fees due on each of the return dates:

Image



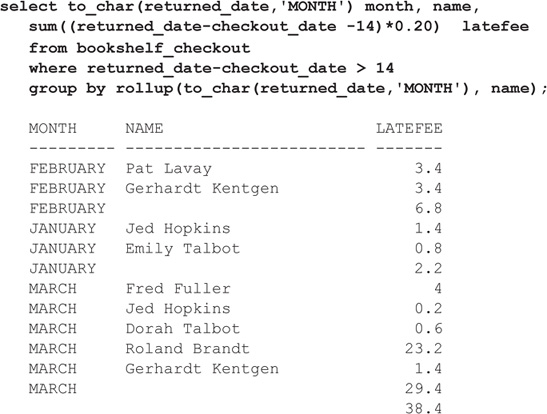
Then we can modify it further to group the late fees by month:

Image



Instead of simply grouping by MONTH and NAME, you can use the ROLLUP function to generate subtotals and totals. In the following example, the GROUP BY clause is modified to include a ROLLUP function call. Notice the additional rows generated at the end of the result set and after each month:

Image



For each month, Oracle has calculated the total late fee and shows it with a NULL value for NAME. The output shows two separate charges of $3.40 in February and a monthly total of $6.80. For the quarter, the total of the late charges is $38.40. You could have calculated these via SQL\*Plus commands (see [Chapter 6](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch06.html#ch06)), but this method allows you to generate these sums via a single SQL command regardless of the tool used to query the database.

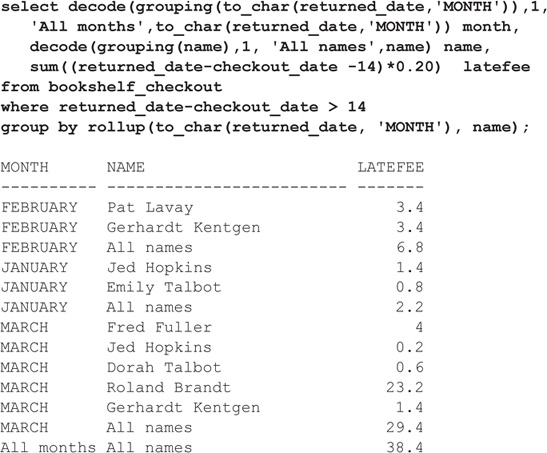
Let’s refine the appearance of the report. You can use the GROUPING function to determine whether the row is a total or subtotal (generated by ROLLUP) or corresponds to a NULL value in the database itself. In the SELECT clause, the NAME column is selected as follows:

Image

Image

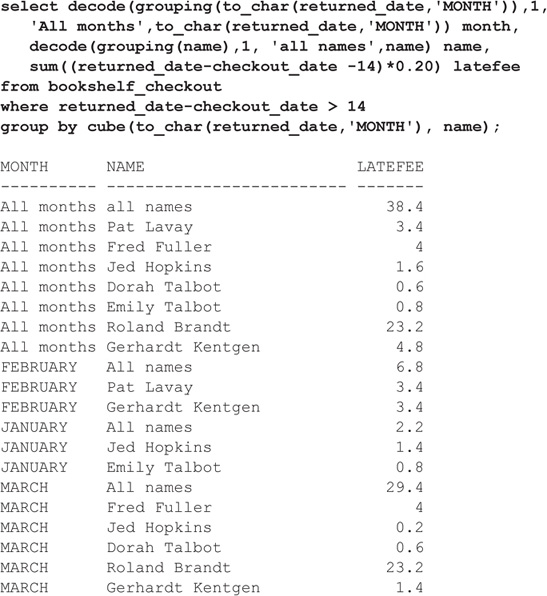
The GROUPING function returns a value of 1 if the column’s value is generated by a ROLLUP action. This query uses DECODE (discussed at length in [Chapter 16](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch16.html#ch16)) to evaluate the result of the GROUPING function. If the GROUPING output is 1, the value was generated by the ROLLUP function, and Oracle prints the phrase ‘All names’; otherwise, it prints the value of the NAME column. We’ll apply similar logic to the MONTH column. The full query is shown in the following listing, along with its output:

Image



You can use the CUBE function to generate subtotals for *all* combinations of the values in the GROUP BY clause. The following query uses the CUBE function to generate these combinations:

Image



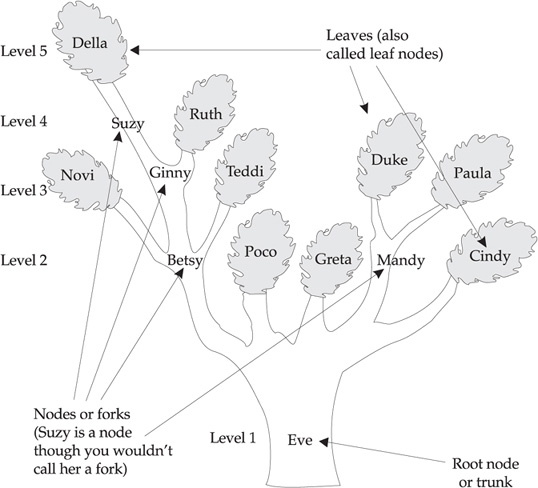
The CUBE function provided the summaries generated by the ROLLUP option, plus it shows the sums by NAME for the “All months” category. Being able to perform these summaries in standard SQL code greatly enhances your ability to pick the best reporting tool for your users.

**Family Trees and CONNECT BY**

One of Oracle’s more interesting but little used or understood facilities is its CONNECT BY clause. Put simply, this method is used to report, in order, the branches of a *family tree.* Such trees are encountered often—the genealogy of human families, livestock, horses; corporate management, company divisions, manufacturing; literature, ideas, evolution, scientific research, theory; and even views built on views.

The CONNECT BY clause provides a means to report on all of the family members in any of these many trees. It lets you exclude branches or individual members of a family tree, and it allows you to travel through the tree either up or down, reporting on the family members encountered during the trip.

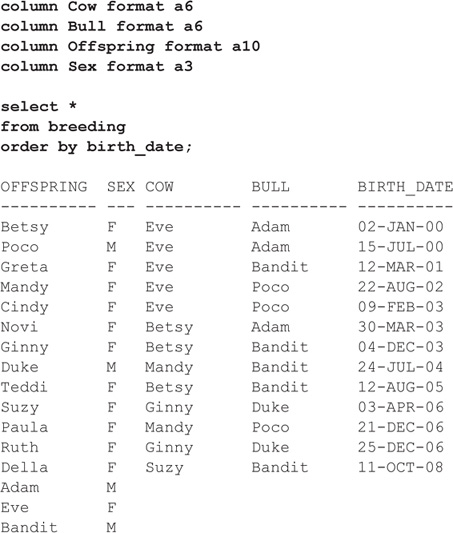
The earliest ancestor in the tree is technically called the *root node.* In everyday English, this is called the *trunk.* Extending from the trunk are *branches,* which have other branches, which have still other branches. The forks where one or more branches split away from a larger branch are called *nodes,* and the very end of a branch is called a *leaf,* or a *leaf node.* [Figure 14-1](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch14.html#ch14fig1) shows a picture of such a tree.



**FIGURE 14-1**. *Eve’s descendants*

The following is a table of cows and bulls born between January 1900 and October 1908. As each offspring is born, it is entered as a row in the table, along with its sex, parents (the cow and bull), and birth date. If you compare the cows and offspring in this table with [Figure 14-1](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch14.html#ch14fig1), you’ll find they correspond. Eve has no recorded cow or bull parent because she was born on a different farm, and Adam and Bandit are bulls brought in for breeding, again with no parents in the table.

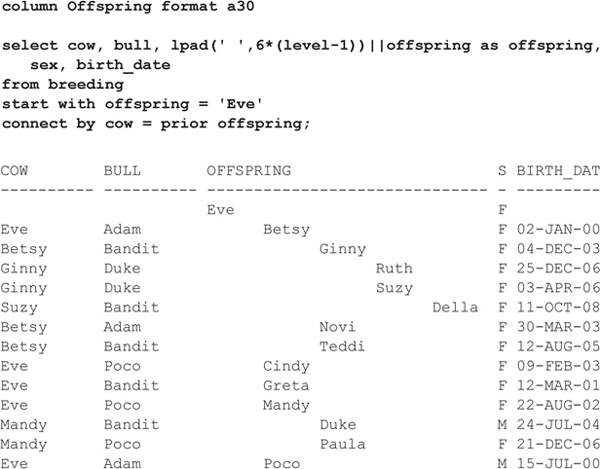
Image



Next, a query is written to illustrate the family relationships visually. You do this using LPAD and a special column, known as a *pseudo-column*, called LEVEL, that you use with CONNECT BY. LEVEL is a number, from 1 for Eve to 5 for Della, that is really the *generation.* If Eve is the first generation of cattle, then Della is the fifth generation. Whenever the CONNECT BY clause is used, the LEVEL column can be used in the SELECT statement to discover the generation of each row. LEVEL is a pseudo-column like SYSDATE and USER. It’s not really a part of the table, but it is available under specific circumstances. The next listing shows an example of using LEVEL.

The results of this query are apparent in the following table, but why did the SELECT statement produce this? How does it work?

Image



Note that this is really [Figure 14-1](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch14.html#ch14fig1) turned clockwise onto its side. Eve isn’t centered, but she is the root node (trunk) of this tree. Her children are Betsy, Cindy, Greta, Mandy, and Poco. Betsy’s children are Ginny, Novi, and Teddi. Ginny’s children are Ruth and Suzy. And Suzy’s child is Della. Mandy also has two children, Duke and Paula.

This tree started with Eve as the first “offspring.” If the SQL statement had said start with Mandy, only Mandy, Duke, and Paula would have been in the output. START WITH defines the beginning of that portion of the tree that will be displayed, and it includes only branches stretching out from the individual that START WITH specifies. START WITH acts just as its name implies.

The LPAD in the SELECT statement is probably somewhat confusing. Recall from [Chapter 7](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch07.html#ch07) the format for LPAD:

Image

Image

That is, take the specified *string* and left-pad it for the specified *length* with the specified *set* of characters. If no *set* is specified, left-pad the string with blanks.

Compare this syntax to the LPAD in the SELECT statement shown earlier:

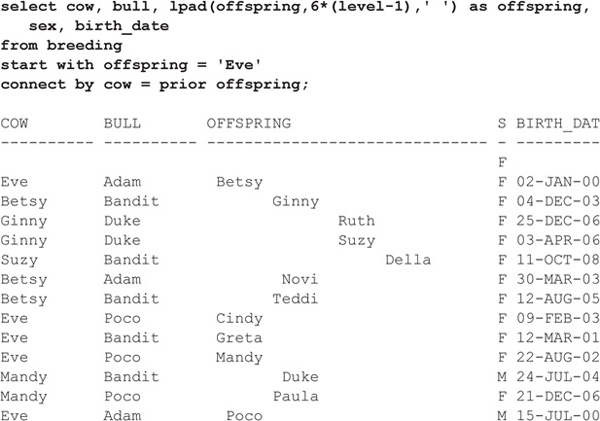
Image

Image

In this case, the *string* is a single character (a space, indicated by the literal space enclosed in single quotation marks). Also, 6\*(LEVEL–1) is the *length*, and because the *set* is not specified, spaces are used. In other words, this tells Oracle to take this string of one space and left-pad it to the number of spaces determined by 6\*(LEVEL–1), a calculation made by first subtracting 1 from the LEVEL and then multiplying this result by 6. For Eve, the LEVEL is 1, so 6\*(1–1), or 0 spaces, is used. For Betsy, the LEVEL (her generation) is 2, so an LPAD of 6 is used. Thus, for each generation after the first, six additional spaces are concatenated to the left of the OFFSPRING column. The effect is obvious in the result just shown. The name of each OFFSPRING is indented by left-padding with the number of spaces corresponding to its LEVEL or generation.

Why is this done, instead of simply applying the LPAD directly to OFFSPRING? There are two reasons. First, a direct LPAD on OFFSPRING would cause the names of the offspring to be right-justified. The names at each level would end up having their last letters lined up vertically. Second, if LEVEL–1 is equal to 0, as it is for Eve, the resulting LPAD of Eve will be 0 characters wide, causing Eve to vanish:

Image



Therefore, to get the proper spacing for each LEVEL, to ensure that EVE appears, and to make the names line up vertically on the left, use LPAD with the concatenation operator or function not directly on the OFFSPRING column.

Now, how does CONNECT BY work? Look again at [Figure 14-1](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch14.html#ch14fig1). Starting with Novi and traveling downward, which cows are the offspring prior to Novi? The first is Betsy, and the offspring just prior to Betsy is Eve. Even though it is not instantly readable, the clause

Image

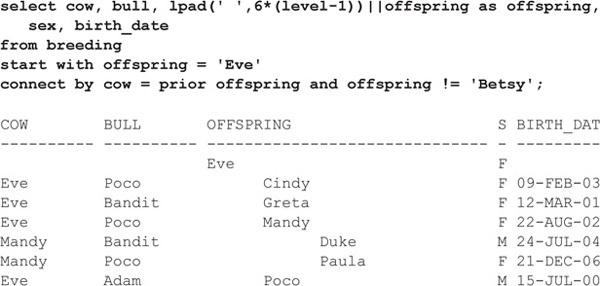
Image

tells Oracle to find the next row in which the value in the COW column is equal to the value in the OFFSPRING column in the prior row. Look at the contents of the BREEDING table and you’ll see that this is true.

**Excluding Individuals and Branches**

There are two methods of excluding cows from a report. One uses the normal WHERE clause technique, and the other uses the CONNECT BY clause itself. The difference is that the exclusion using the CONNECT BY clause excludes not just the cow mentioned, but all of its children as well. If you use CONNECT BY to exclude Betsy, then Novi, Ginny, Teddi, Suzy, Ruth, and Della all vanish. The CONNECT BY clause really tracks the tree structure. If Betsy had never been born, none of her offspring would have been either. In this example, the AND clause modifies the CONNECT BY clause:

Image



The WHERE clause removes only the cow or cows it mentions. If Betsy dies, she is removed from the chart, but her offspring are not. In fact, notice that Betsy is still there under the COW column as the mother of her children, Novi, Ginny, and Teddi:

Image

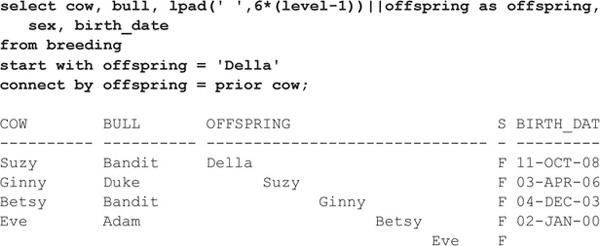


The order in which the family tree is displayed when using CONNECT BY is basically level by level, left to right, as shown in [Figure 14-1](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch14.html#ch14fig1), starting with the lowest level, Level 1.

**Traveling Toward the Roots**

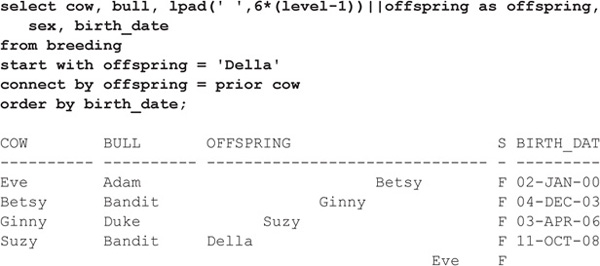
Thus far, the direction of travel in reporting on the family tree has been from parents toward children. Is it possible to start with a child and move backward to parent, grandparent, great-grandparent, and so on? To do so, the word PRIOR is simply moved to the other side of the equal sign. In the following examples, the OFFSPRING is set equal to the PRIOR COW value; in the earlier examples, the COW was set equal to the PRIOR OFFSPRING value. The following traces Della’s ancestry:

Image



This shows Della’s own roots, but it’s a bit confusing if compared to the previous displays. It looks as if Della is the ancestor and Eve the great-great-granddaughter. Adding an ORDER BY for BIRTH\_DATE helps, but Eve is still further to the right:

Image



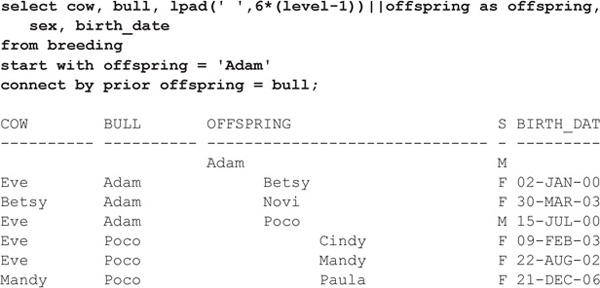
The solution is simply to change the calculation in the LPAD function call:

Image



Finally, look how different this report is when the CONNECT BY tracks the parentage of the bull. Here are Adam’s offspring:

Image



Adam and Bandit were the original bulls at the initiation of the herd. To create a single tree that reports both Adam’s and Bandit’s offspring, you would have to invent a “father” for the two of them, which would be the root of the tree. One of the advantages that these alternative trees have over the type of tree shown earlier is that many inheritance groups—from families to projects to divisions within companies—can be accurately portrayed in more than one way:

Image

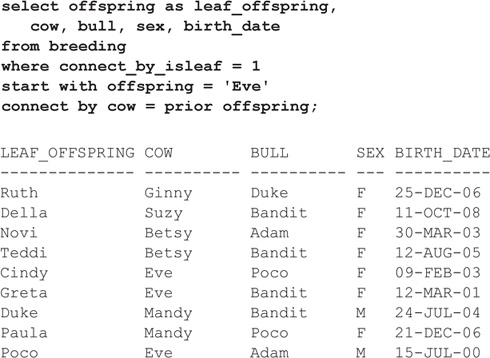


**Finding the Leaves**

Occasionally it can be useful to identify those offspring who have themselves not produced any offspring, in other words, identifying only the leaves of the tree. Another pseudo-column, new to Oracle Database 11*g*, is CONNECT\_BY\_ISLEAF. It has a value of 1 if this row is a leaf node in the family tree.

Thus, we can take one of the earlier queries that shows the entire family tree and filter it to include only rows with CONNECT\_BY\_ISLEAF=1:

Image



Therefore, nine offspring in the herd have not yet been parents.

**The Basic Rules**

Using CONNECT BY and START WITH to create tree-like reports is not difficult, but you must follow certain basic rules:

image The order of the clauses when using CONNECT BY is as follows:

**1**. SELECT

**2**. FROM

**3**. WHERE

**4**. START WITH

**5**. CONNECT BY

**6**. ORDER BY

image PRIOR forces reporting to be from the root out toward the leaves (if the PRIOR column is the parent) or from a leaf toward the root (if the PRIOR column is the child).

image A WHERE clause eliminates individual rows from the tree, but not their descendants (or ancestors, if PRIOR is on the right side of the equal sign).

image A qualification in the CONNECT BY (particularly a not equal) eliminates both an individual row and all of its descendants (or ancestors, depending on how you trace the tree).

image CONNECT BY cannot be used with a table join in the WHERE clause in versions previous to Oracle 11*g*.

image Pseudo-columns such as LEVEL and CONNECT\_BY\_ISLEAF can help to prune the tree when only certain rows need to be in the query result.

This particular set of commands is one that few people are likely to remember correctly. However, with a basic understanding of trees and inheritance, you should be able to construct a proper SELECT statement to report on a tree just by referring to this chapter for correct syntax and examples.

#### ****Creating a View****

Since you’ve already seen the techniques for creating a view in prior chapters, they are not reviewed extensively here. However, this section gives several additional points about views that will prove useful.

View stability maintains the accessibility of a view even if the view is re-created or new columns are added to an underlying table. Existing code that references the view does not have to change right away unless the existing code needs the new columns.

If a view is based on a single underlying table, you can INSERT, UPDATE, or DELETE rows in the view. This will actually INSERT, UPDATE, or DELETE rows in the underlying table. There are restrictions on your ability to do this, although the restrictions are quite sensible:

image You cannot INSERT if the underlying table has any NOT NULL columns that don’t appear in the view.

image You cannot INSERT or UPDATE if any one of the view’s columns referenced in the INSERT or UPDATE contains functions or calculations.

image You cannot INSERT, UPDATE, or DELETE if the view contains GROUP BY, DISTINCT, or a reference to the pseudo-column ROWNUM.

You can INSERT into a view based on multiple tables if Oracle can determine the proper rows to insert into the correct table. In a multi-table view, Oracle determines which of the tables are key-preserved. If a view contains enough columns from a table to identify the primary key for that table, the key is preserved, and Oracle may be able to insert rows into the table via the view.

##### ****View Stability****

Remember that the results of querying a view are built instantly from a table (or tables) when you execute the query. Until that moment, the view has no data of its own, as a table does. It is merely a description (a SQL statement) of what information to pull out of other tables and how to organize it. As a consequence, if a table is dropped, the validity of a view is destroyed. Attempting to query a view where the underlying table has been dropped produces an error message about the view.

Image

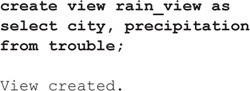
**NOTE**

The sole exception to this rule is the use of materialized views. In fact, a materialized view is actually a table that stores data you would normally query via a view. Materialized views are described in detail in [*Chapter 26*](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch26.html#ch26).

In the following sequence, a view is created on an existing table, the table is dropped, and the view then is queried.

First, the view is created:

Image



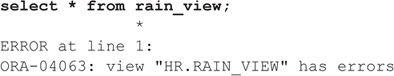
The underlying table is dropped:

Image

Image

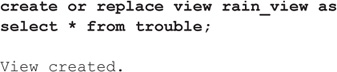
The view is queried:

Image



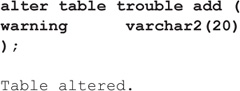
Similarly, a view is created using the asterisk in the view creation:

Image



But then the underlying table is altered:

Image



Despite the change to the view’s base table, the view is still valid, but the WARNING column is not visible through it. After altering the table, replace the view so the WARNING column is visible. To re-create a view while keeping in place all the privileges that have been granted for it, use the CREATE OR REPLACE VIEW command, as shown in the following listing. This command replaces the view text of an existing view with the new view text, leaving the old grants on the view unaffected.

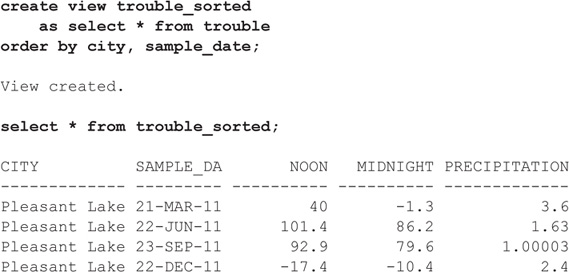
Image

Image

##### ****Using ORDER BY in Views****

You can use an ORDER BY in a CREATE VIEW statement, as shown in the following listing based on the re-created TROUBLE table (note that queries against this view may incur a performance penalty because of the additional sort activity they perform):

Image



##### ****Creating a Read-Only View****

You can use the WITH READ ONLY clause of the CREATE VIEW command to prevent users from manipulating records via the view. Consider the RAIN\_VIEW view created in the previous section:

Image

Image

If a user has the ability to DELETE records from the TROUBLE table, the user could DELETE the TROUBLE records via the RAIN\_VIEW view:

Image

Image

The user could also INSERT or UPDATE the records in the TROUBLE table by performing those operations against the RAIN\_VIEW view. If the view is based on a join of multiple tables, the user’s ability to UPDATE the view’s records is limited; a view’s base tables cannot be updated unless only one table is involved in the UPDATE and the updated table’s full primary key is included in the view’s columns.

To prevent modifications to the base tables via a view, you can use the WITH READ ONLY clause of the CREATE VIEW command. If you use the WITH READ ONLY clause when creating a view, users will only be able to SELECT records from the view. Users will be unable to manipulate the records obtained from the view even if the view is based on a single table:

Image

Image

You can also use INSTEAD OF triggers to manage the data-manipulation commands executed against views. See [Chapter 38](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch38.html#ch38) for details on INSTEAD OF triggers.

#### ****Indexes****

An index is a simple concept. It is typically a listing of keywords accompanied by the location of information on a subject. To find information on indexes, for instance, you look up the word “indexes” in the index at the back of this book. It will give the number of the page you are reading now. The word “indexes” is the key, and the page numbers given point you to the location of discussions about indexes in this book.

Although you could find the information on indexes simply by reading through this book until you encountered the page with the information on it, this method would be slow and time-consuming. Because the index at the back of the book is in alphabetical order, you can quickly go to the appropriate spot in the index (without reading every entry) where “index” is found. If the word is not common, this is quicker than reading through the book from front to back. These same principles apply to Oracle indexes. For example, if you are searching for a particular book, you can query with a limiting condition on the TITLE column:

Image

Image

If BOOKSHELF does not have an index on the PUBLISHER column, Oracle has to read every row in the table until it finds all publishers that match the WHERE clause of your query. If the table is small, that may not cause a performance problem. As the table grows in size, the time required to return all the matching rows to the user may impact the application’s performance and the business process it supports.

To speed data retrieval, you can create an index on the PUBLISHER column. Then, when you execute the same query, Oracle first looks in the index, which is sorted, thus finding the publisher named “Scholastic” very quickly (Oracle doesn’t read every entry, but jumps directly within close vicinity of the name, much as you would in looking through the index of a book). The index entry then gives Oracle the exact location in the table (and on disk) of the row(s) for that publisher. The standard type of index in Oracle is called a B\*-tree index, matching column values to their related ROWIDs.

Indexing an important column (one that’s likely to appear in a WHERE clause) generally speeds up Oracle’s response to a query. Indexing likewise speeds up queries where two tables are joined, if the columns that are related (by the WHERE clause) are indexed. These are the basics of indexing; the rest of this section shows a number of additional features and issues related to indexing that affect how quickly it works. For the impact of indexes on the optimization of your queries, see [Chapter 46](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch46.html#ch46).

##### ****Creating an Index****

You create an index via the CREATE INDEX command. When you designate a primary key or a unique column during table creation or maintenance, Oracle automatically creates a unique index to support that constraint. The full command syntax for CREATE INDEX is shown in the Alphabetical Reference. Its most commonly used format is as follows:

Image

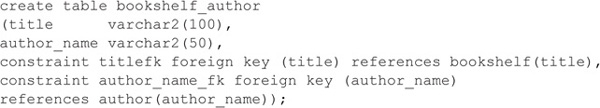
Image

index must be a unique name and follow the naming conventions of Oracle columns. table is simply the name of the table on which the index will be established, and column is the name of the column being indexed.

Bitmap indexes allow you to create useful indexes on columns with very few distinct values; see “Creating a Bitmap Index,” later in this chapter. The REVERSE keyword tells Oracle to reverse the bytes of the indexed value, which may improve the I/O distribution during the insert of many sequential data values.

You can establish a single index on multiple columns by listing the columns one after the other, separated by commas. For the following examples, the BOOKSHELF\_AUTHOR table was created without a primary key:

Image



##### ****Enforcing Uniqueness****

Recall from [Part I](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch00_part01.html#ch00_part01) that a set of tables is said to be in Third Normal Form if all the columns in each table’s rows are dependent only on the primary key. In the BOOKSHELF\_AUTHOR table, the primary key is the combination of the columns TITLE and AUTHOR\_NAME. In other tables, a primary key might be an employee ID, a client ID, an account number, or, in a bank, a combination of branch number and account number.

In each of these cases, the uniqueness of the primary key is critical. A bank with duplicate account numbers, or a billing system with duplicate client IDs, would wreak havoc as transactions were posted to accounts belonging to different people but having the same primary key (this is why names generally are not commonly used as primary keys—there are too many duplicates). To avoid this danger, have your database help prevent the creation of duplicate primary keys. Oracle offers two facilities that help:

image You can guarantee the uniqueness of a key through either indexing or constraints.

image You can use the sequence generators (discussed later in this chapter).

##### ****Creating a Unique Index****

You can create a unique index on the combination of TITLE and AUTHOR\_NAME in the BOOKSHELF\_AUTHOR table in one of three ways: by creating a primary key constraint, by creating a unique constraint, or by creating a unique index. If you create a constraint, you will be able to create foreign keys that refer to it. If you create the unique index first, you will still be able to create a primary key on the table—Oracle will use the existing index as the primary key index.

The following listing shows the CREATE INDEX command for this multicolumn index:

Image

Image

The primary key method is shown here:

Image

Image

To create a unique constraint, just replace the PRIMARY KEY clause with UNIQUE. When you attempt to create a unique index on a table that already has data in it, the command will fail if any duplicates exist. If the CREATE UNIQUE INDEX statement succeeds, any future attempt to INSERT (or UPDATE) a row that would create a duplicate key will fail and result in this error message:

Image

Image

When you create an index, it requires storage space. See “Placing an Index in the Database,” later in this chapter, for details concerning the location of the created indexes.

##### ****Creating a Bitmap Index****

To help tune queries that use nonselective columns in their limiting conditions, you can use bitmap indexes. Bitmap indexes should only be used if the data is infrequently updated because they add to the cost of all data-manipulation transactions against the tables they index.

Image

**NOTE**

Bitmap indexes should not be used for tables involved in online transaction-processing applications because of the internal mechanisms Oracle uses to maintain them. Restrict their usage to tables involved in batch transactions.

Bitmap indexes are appropriate when nonselective columns are used as limiting conditions in a query. For example, if there are very few distinct RATING values in a very large BOOKSHELF table, you would not usually create a traditional B\*-tree index on RATING, even if it is commonly used in WHERE clauses. However, RATING may be able to take advantage of a bitmap index.

Internally, a bitmap index maps the distinct values for the columns to each record. For this example, assume there are five RATING values (1, 2, 3, 4, and 5) in a very large BOOKSHELF table. Because there are five RATING values, there are five separate bitmap entries for the RATING bitmap index. If the first five rows in the table have a RATING value of 1, and the next five have a RATING value of 2, then the RATING bitmap entries would resemble those shown in the following listing:

Image



In the preceding listing, each column of 0s and 1s represents a row in the BOOKSHELF table. Because ten rows are considered, ten bitmap values are shown. Reading the bitmap for RATING, the first five records have a RATING value of 1 (the “1” values), and the next five do not (the “0” values). There is a separate bitmap entry for each possible value.

The Oracle Optimizer can dynamically convert bitmap index entries to ROWIDs during query processing. This conversion capability allows the optimizer to use indexes on columns that have many distinct values (via B\*-tree indexes) and on those that have few distinct values (via bitmap indexes).

To create a bitmap index, use the BITMAP clause of the CREATE INDEX command, as shown in the following listing. You should indicate its nature as a bitmap index within the index name so it will be easy to detect during tuning operations.

Image

Image

If you choose to use bitmap indexes, you need to weigh the performance benefit during queries against the performance cost during data-manipulation commands. The more bitmap indexes there are on a table, the greater the cost will be during each transaction. You should not use bitmap indexes on a column that frequently has new values added to it. Each addition of a new value to the RATING column requires that a corresponding new bitmap be created.

##### ****When to Create an Index****

Indexes are most useful on larger tables, on columns that are likely to appear in WHERE clauses either as a simple equality, such as

Image

Image

or in joins, such as this:

Image

Image

In addition to supporting WHERE clauses and joins, indexes also support ORDER BY clauses and the MAX and MIN functions. In some circumstances, Oracle may choose to scan an index instead of performing a full scan on the table. See [Chapter 46](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch46.html#ch46) for details on the Oracle Optimizer’s use of indexes.

##### ****Creating Invisible Indexes****

The presence of a new index may alter how an existing application works. In most cases, this is the intended behavior: You are creating the index to change the execution path chosen by a query. The new index should be designed to support the known query execution paths, and it will slow the performance of INSERTs and other DML operations. As of Oracle 11g, you can introduce an index to your environment in stages, first creating it and then making it visible to the optimizer.

After creating an index, you can tell the optimizer to not consider it when generating execution plans via the ALTER INDEX command:

Image

Image

With the index in an “invisible” state, you can now test your commands to determine exactly how the index will impact your operations. The invisible index will only be used if you use an INDEX hint (see [Chapter 46](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch46.html#ch46)) that explicitly names the index. The index will not be considered by the optimizer for general usage while it is invisible. Select the VISIBILITY column from the USER\_INDEXES data dictionary view to determine if an index has been set to be invisible to the optimizer. Regardless of its visibility status, the index continues to use space and impact DML operations.

##### ****Variety in Indexed Columns****

Traditional (B\*-tree) indexes are most useful on columns with a significant amount of variety in their data. For instance, a column that indicates whether a company is a current client with a Y or N value would be a poor choice for a traditional index and could actually slow down a query; a bitmap index would be a much better choice for that example. A telephone number column would be a good candidate for a B\*-tree index. An area code column would be marginal, depending on the distribution of unique area code values in the table.

In a multicolumn index, put the column likely to be accessed most often first. You can use the optimizer’s skip-scan feature to use multicolumn indexes even if the leading column is not mentioned in the query. See [Chapter 46](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch46.html#ch46) for examples of index usage.

Small tables may be better left unindexed, except to enforce uniqueness in the primary key. A small table is one that takes so few database blocks that Oracle can read all its data with one physical read; beyond that, indexing will nearly always be productive. The number of blocks read during a physical read is set via the DB\_FILE\_MULTIBLOCK\_READ\_COUNT initialization parameter.

Bitmap indexes present a viable indexing alternative for columns that have very few distinct values. Bitmap indexes are commonly used for “flag” columns that are restricted to values such as Y and N. Bitmap indexes are particularly effective when multiple bitmap indexes are used in a query; the optimizer can quickly evaluate the bitmaps and determine which rows meet all the criteria for which bitmap indexes are available.

##### ****How Many Indexes to Use on a Table****

You can create many indexes on a single table, with many columns in each index. The tradeoff for indexing too many columns is the lessened speed of inserting new rows: Every index must be updated whenever an INSERT, UPDATE, or DELETE occurs. If your table is used primarily for queries, the only cost of indexing as many columns as you can (that have variety and are used in WHERE clauses, of course) is the extra disk space used. Strictly from a data-loading performance perspective, it is better to have a small number of indexes with many columns than to have many indexes with a small number of columns in each.

Except in cluster indexes (discussed later in this chapter), column values that are NULL do not appear in an index. Indexes based on more than one column have an entry if any of the columns are not NULL. If all indexed columns are NULL for a given row, no entry appears for this row in the index.

##### ****Placing an Index in the Database****

You can specify where the index to a table is placed by assigning it to a specific tablespace. As will be described in detail in [Chapter 22](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch22.html#ch22), a tablespace is a logical division of the database, corresponding to one or more datafiles. Datafiles provide the physical storage for the database—sections of disk where tables and indexes are stored. Databases have several tablespaces, each with its own name. To enhance manageability and sometimes performance (if you’re not using ASM), an index for a table can be placed in a tablespace that is on a physically separate disk drive from its corresponding table.

To specify the tablespace in which to locate an index, the normal CREATE INDEX statement is simply followed by the word TABLESPACE and the tablespace name, as shown here:

Image

Image

In this example, BA\_INDEXES is the name given to a tablespace previously created by the database administrator. The use of the TABLESPACE option in a CREATE INDEX statement lets you physically separate your tables from their associated indexes.

When you create a primary key or unique constraint, Oracle automatically creates an index to enforce uniqueness. Unless you specify otherwise, that index is created in the same tablespace as the table that the constraint modifies, and it uses the default storage parameters for that tablespace. Because that storage location is often not ideal, you should take advantage of the USING INDEX clause when creating primary key and unique constraints.

The USING INDEX clause allows you to specify the storage parameters and tablespace location for an index created by a constraint. In the following example, a primary key is created on the BOOKSHELF\_AUTHOR table and is placed in the BA\_INDEXES tablespace. This example assumes that there is not an existing index on the specified columns.

Image

Image

See “Integrity Constraint” in the Alphabetical Reference for further options for the USING INDEX clause, and see CREATE INDEX in the Alphabetical Reference for performance-related index-creation options.

##### ****Rebuilding an Index****

Oracle provides a fast index rebuild capability that allows you to re-create an index without having to drop the existing index. The currently available index is used as the data source for the index, instead of the table’s being used as the data source. During the index rebuild, you can change its STORAGE parameters and TABLESPACE assignment.

In the following example, the BA\_PK index is rebuilt (via the REBUILD clause). Its storage parameters are changed to use an initial extent size of 8MB and a next extent size of 4MB in the BA\_INDEXES tablespace.

Image

Image

Image

**NOTE**

When the BA\_PK index is rebuilt, there must be enough space for both the old index and the new index to exist simultaneously. After the new index has been created, the old index will be dropped.

When you create an index that is based on previously indexed columns, Oracle may be able to use the existing indexes as data sources for the new index. The Oracle Optimizer may use parts of existing composite indexes as needed for queries, so you may not need to create many indexes to support the most common queries.

You can rebuild indexes while they are being accessed via the REBUILD ONLINE clause of the ALTER INDEX command.

Image

Image

##### ****Function-Based Indexes****

You can create function-based indexes. Any query that performs a function on a column generally does not use that column’s index. Therefore, this query could not use an index on the TITLE column:

Image

Image

However, this query could because it does not perform the UPPER function on the TITLE column:

Image

Image

You can create indexes that allow function-based accesses to be supported by index accesses. Instead of creating an index on the column TITLE, you can create an index on the column expression UPPER(TITLE), as shown in the following listing:

Image

Image

Although function-based indexes can be useful, be sure to consider the following questions when creating them:

image Can you restrict the functions that will be used on the column? If so, can you restrict all functions from being performed on the column?

image Do you have adequate storage space for the additional indexes?

image When you drop the table, will you be dropping more indexes (and, therefore, more extents) than before? How will that impact the time required to drop the table?

Function-based indexes are useful, but you should implement them sparingly. The more indexes you create on a table, the longer all INSERT, UPDATE, and DELETE operations take. Typically, each index adds three times the overhead of maintaining a row without an index.

#### ****Clusters****

Clustering is a method of storing tables that are intimately related and often joined together into the same area on disk. For example, instead of the BOOKSHELF table being in one section of the disk and the BOOKSHELF\_AUTHOR table being somewhere else, their rows could be interleaved together in a single area, called a cluster. The cluster key is the column or columns by which the tables are usually joined in a query (for example, TITLE for the BOOKSHELF and BOOKSHELF\_AUTHOR tables). To cluster tables, you must own the tables you are going to cluster together.

The following is the basic format of the CREATE CLUSTER command:

Image

Image

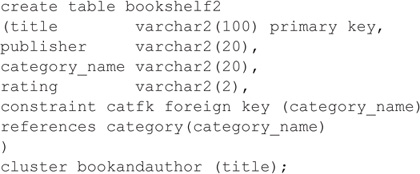
The clustername follows table-naming conventions, and column datatype is the name and datatype you will use as the cluster key. The column name may be the same as one of the columns of a table you will put in this cluster, or it may be any other valid name. Here’s an example:

Image

Image

This creates a cluster (disk space is set aside, as it would be for a table) with nothing in it. The use of COL1 for the cluster key is irrelevant; you’ll never use it again. However, its definition should match the primary key of the table to be added. Next, tables are created to be included in this cluster:

Image



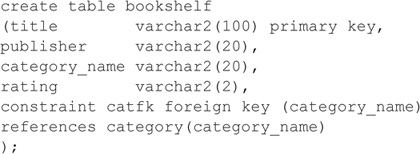
Prior to inserting rows into BOOKSHELF, you must create a cluster index:

Image

Image

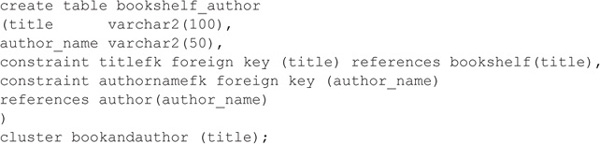
Recall that the presence of a CLUSTER clause here precludes the use of a TABLESPACE or STORAGE clause. Note how this structure differs from a standard CREATE TABLE statement:

Image



In the first CREATE TABLE statement, the cluster BOOKANDAUTHOR (TITLE) clause follows the closing parenthesis of the list of columns being created in the table. BOOKANDAUTHOR is the name of the cluster previously created. TITLE is the column in this table that will be stored in the cluster key COL1. It is possible to have multiple cluster keys in the CREATE CLUSTER statement and to have multiple columns stored in those keys in the CREATE TABLE statement. Notice that nowhere does either statement say explicitly that the TITLE column goes into the COL1 cluster key. The match-up is done by position only: COL1 and TITLE were both the first objects mentioned in their respective cluster statements. Multiple columns and cluster keys are matched first to first, second to second, third to third, and so on. Now a second table is added to the cluster:

Image



When these two tables are clustered, each unique title is actually stored only once, in the cluster key. The columns from both of these tables are attached to each title. The data from both of these tables is actually stored in a single location, almost as if the cluster were a big table containing data drawn from both of the tables that make it up.

An additional cluster option, a hash cluster, uses the cluster column values to determine the physical location in which the row is stored. See the entry for the CREATE CLUSTER command in the Alphabetical Reference.

#### ****Sequences****

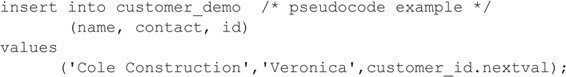
You can assign unique numbers, such as customer IDs, to columns in your database by using a sequence; you don’t need to create a special table and code to keep track of the unique numbers in use. You do this by using the CREATE SEQUENCE command, as shown here:

Image

Image

This creates a sequence that can be accessed during INSERT and UPDATE commands (also SELECT, although this is rare). Typically, the unique sequence value is created with a statement like the following:

Image



The NEXTVAL attached to CUSTOMER\_ID tells Oracle you want the next available sequence number from the CUSTOMER\_ID sequence. This is guaranteed to be unique; Oracle will not give it to anyone else. To use the same number more than once (such as in a series of INSERTs into related tables), CURRVAL is used instead of NEXTVAL, after the first use. That is, using NEXTVAL ensures that the sequence table gets incremented and that you get a unique number, so you have to use NEXTVAL first. Once you’ve used NEXTVAL, that number is stored in CURRVAL for your use anywhere—until you use NEXTVAL again, at which point both NEXTVAL and CURRVAL change to the new sequence number.

If you use both NEXTVAL and CURRVAL in a single SQL statement, both will contain the value retrieved by NEXTVAL. Neither of these can be used in subqueries, as columns in the SELECT clause of a view, with DISTINCT, UNION, INTERSECT, or MINUS, or in the ORDER BY, GROUP BY, or HAVING clause of a SELECT statement.

You can also cache sequence values in memory for faster access, and you can make the sequence cycle back to its starting value once a maximum value is reached. In RAC environments, Oracle recommends caching 20,000 sequence values per instance to avoid contention during writes. For non-RAC environments, you should cache at least 1,000 values. Note that if you flush the shared pool portion of the instance or you shut down and restart the database, any cached sequence values will be lost and there will be gaps in the sequence numbers stored in the database. See CREATE SEQUENCE in the Alphabetical Reference.

### ****USING MATERIALIZED VIEWS****

To improve the performance of an application, you can make local copies of remote tables that use distributed data or create summary tables based on GROUP BY operations. Oracle provides materialized views to store copies of data or aggregations. Materialized views can be used to replicate all or part of a single table or to replicate the result of a query against multiple tables; the database can automatically refresh the replicated data at time intervals that you specify. In this chapter, you will learn about the general usage of materia44lized views, including their refresh strategies; this is followed by a discussion of the optimization strategies available.

#### ****Functionality****

Materialized views are copies (also known as replicas or snapshots) of data based on queries. In its simplest form, a materialized view is similar to a table created by a command such as the following:

Image

Image

In this example, a regular table named LOCAL\_BOOKSHELF is created in the local database and is populated with data from a remote database (defined by the database link named REMOTE\_ CONNECT). Once the LOCAL\_BOOKSHELF table is created, though, its data may immediately become out of sync with the master table (BOOKSHELF@REMOTE\_CONNECT). Also, LOCAL\_ BOOKSHELF may be updated by local users, further complicating its synchronization with the master table.

Despite these synchronization problems, there are benefits to replicating data in this way. Creating local copies of remote data may improve the performance of distributed queries, particularly if the master table’s data does not change frequently. You may also use the local table creation process to restrict the rows returned, restrict the columns returned, or generate new columns (such as by applying functions to selected values). This strategy is common in decision-support environments, in which complex queries are used to periodically “roll up” data into summary tables for use during analyses.

Materialized views automate the data replication and refresh processes. When materialized views are created, a refresh interval is established to schedule refreshes of replicated data. Local updates can be prevented, and transaction-based refreshes can be used. Transaction-based refreshes, available for many types of materialized views, send from the master database only those rows that have changed for the materialized view. This capability, described later in this chapter, may significantly improve the performance of your refreshes.

#### ****Required System Privileges****

To create a materialized view, you must have the privileges needed to create or access the underlying objects it will use. You must have the CREATE MATERIALIZED VIEW privilege, as well as the CREATE TABLE or CREATE ANY TABLE system privilege. In addition, you must have either the UNLIMITED TABLESPACE system privilege or a sufficient specified space quota in a local tablespace. To create a refresh-on-commit materialized view, you must also have the ON COMMIT REFRESH system privilege on any tables you do not own or the ON COMMIT REFRESH system privilege.

Materialized views of remote tables require queries of remote tables; therefore, you must have privileges to use a database link that accesses the remote database. The link you use can be either public or private. If the database link is private, you need to have the CREATE DATABASE LINK system privilege to create the database link. See [Chapter 25](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch25.html#ch25) for further information on database links.

If you are creating materialized views to take advantage of the query rewrite feature (in which the optimizer dynamically chooses to select data from the materialized view instead of the underlying table), you must have the QUERY REWRITE privilege. If the tables are in another user’s schema, you must have the GLOBAL QUERY REWRITE privilege. If the materialized view is created with ON COMMIT REFRESH specified, you must have the ON COMMIT REFRESH system privilege or the ON COMMIT REFRESH object privilege on each table outside your schema.

Image

**NOTE**

As of Oracle Database 11g, queries that reference remote tables can support query rewrite.

In Oracle Database 11g, query rewrite has been enhanced to support queries containing inline views. Prior to this release, queries containing inline views could rewrite only if there was an exact text match with the inline views in the materialized views.

For materialized views in Oracle Database 12c, availability has been improved. By leveraging partitioning and the known dependencies between base tables and related materialized views, the materialized view can be refreshed simultaneously with the base table(s). As a result, the amount of time that a materialized view is stale is significantly reduced.

#### ****Required Table Privileges****

When creating a materialized view, you can reference tables in a remote database via a database link. The account that the database link uses in the remote database must have access to the tables and views used by the database link. You cannot create a materialized view based on objects owned by the user SYS.

Within the local database, you can grant the SELECT privilege on a materialized view to other local users. Because most materialized views are read-only (although they can be updatable), no additional GRANTs are necessary. If you create an updatable materialized view, you must grant users the UPDATE privilege on both the materialized view and the underlying local table it accesses.

#### ****Read-Only vs. Updatable****

A read-only materialized view cannot pass data changes from itself back to its master table. An updatable materialized view can send changes back to its master table.

Although that may seem to be a simple distinction, the underlying differences between these two types of materialized views are not simple. A read-only materialized view is implemented as a CREATE TABLE AS SELECT command. When transactions occur, they occur only within the master table; the transactions are optionally sent to the read-only materialized view. Thus, the method by which the rows in the materialized view change is controlled—the materialized view’s rows only change following a change to the materialized view’s master table.

In an updatable materialized view, you have less control over the method by which rows in the materialized view are changed. Rows may be changed based on changes in the master table, or rows may be changed directly by users of the materialized view. As a result, you need to send records from the master table to the materialized view, and vice versa. Because multiple sources of changes exist, multiple masters exist (referred to as a multimaster configuration).

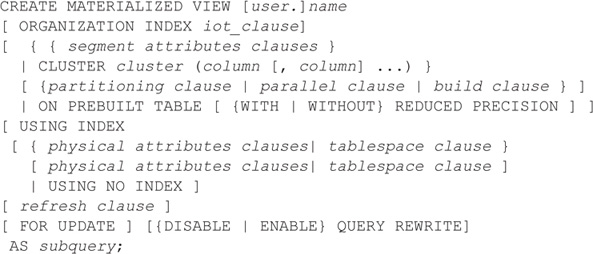
During the transfer of records from the materialized view to master, you need to decide how you will reconcile conflicts. For example, what if the record with ID=1 is deleted at the materialized view site, while at the master site, a record is created in a separate table that references (via a foreign key) the ID=1 record? You cannot delete the ID=1 record from the master site because that record has a “child” record that relates to it. You cannot insert the child record at the materialized view site because the parent (ID=1) record has been deleted. How do you plan to resolve such conflicts?

Read-only materialized views let you avoid conflict resolution by forcing all transactions to occur in the controlled master table. This may limit your functionality, but it is an appropriate solution for the vast majority of replication needs. If you need multimaster replication, see the Advanced Replication Guidefor guidelines and detailed implementation instructions.

#### ****CREATE MATERIALIZED VIEW Syntax****

The basic syntax for creating a materialized view is shown in the following listing. See the Alphabetical Reference for the full command syntax. Following the command description, examples are given that illustrate the creation of local replicas of remote data.

Image



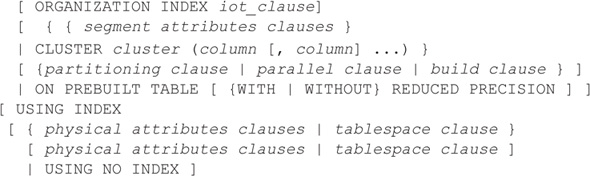
The CREATE MATERIALIZED VIEW command has four major sections. The first section is the header, in which the materialized view is named (the first line in the listing):

Image

Image

The materialized view will be created in your user account (schema) unless a different schema name is specified in the header. In the second section, the storage parameters are set:

Image



The storage parameters will be applied to a table that will be created in the local database. For information about the available storage parameters, see the STORAGE entry in the Alphabetical Reference. If the data has already been replicated to a local table, you can use the ON PREBUILT TABLE clause to tell Oracle to use that table as a materialized view.

Image

**NOTE**

You can specify the storage parameters to be used for the index that is automatically created on the materialized view.

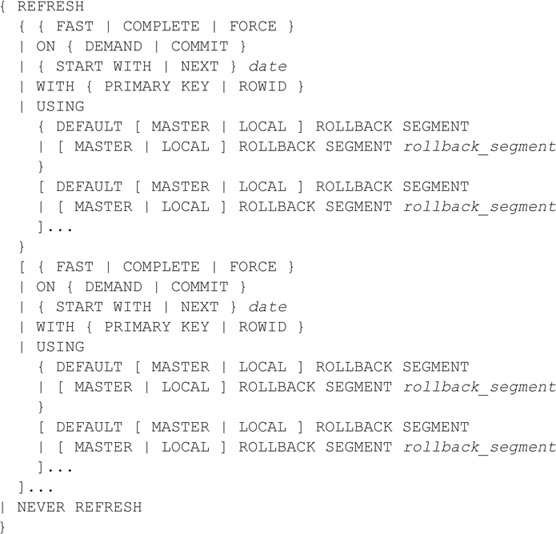
In the third section, the refresh options are set:

Image

Image

The syntax for the refresh clause is

Image



The REFRESH option specifies the mechanism Oracle should use when refreshing the materialized view. The three options available are FAST, COMPLETE, and FORCE. FAST refreshes are only available if Oracle can match rows in the materialized view directly to rows in the base table(s); the materialized view refresh process uses tables called materialized view logs to send specific rows from the master table to the materialized view. COMPLETE refreshes truncate the data and re-execute the materialized view’s base query to repopulate it. FORCE refreshes tell Oracle to use a FAST refresh if it is available; otherwise, a COMPLETE refresh is used. If you have created a simple materialized view but want to use complete refreshes, specify REFRESH COMPLETE in your CREATE MATERIALIZED VIEW command. The refresh options are further described in “Refreshing Materialized Views,” later in this chapter.

Within this section of the CREATE MATERIALIZED VIEW command, you also specify the mechanism used to relate values in the materialized view to the master table—whether ROWIDs or primary key values should be used. By default, primary keys are used.

If the master query for the materialized view references a join or a single-table aggregate, you can use the ON COMMIT option to control the replication of changes. If you use ON COMMIT, changes are sent from the master to the replica when the changes are committed on the master table. If you specify ON DEMAND, the refresh occurs when you manually execute a refresh command.

The fourth section of the CREATE MATERIALIZED VIEW command is the query that the materialized view uses:

Image

Image

If you specify FOR UPDATE, the materialized view is updatable; otherwise, it is read-only. Most materialized views are read-only replicas of the master data. If you use updatable materialized views, you need to be concerned with issues such as two-way replication of changes and the reconciliation of conflicting data changes. Updatable materialized views are an example of multimaster replication; for full details on implementing a multimaster replication environment, see the Advanced Replication Guide.

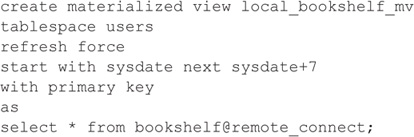
Image

**NOTE**

The query that forms the basis of the materialized view should not use the USER or SYSDATE pseudo-columns.

The following example creates a read-only materialized view called LOCAL\_BOOKSHELF in a local database, based on a remote table named BOOKSHELF that is accessible via the REMOTE\_CONNECT database link. The materialized view is placed in the USERS tablespace.

Image



Oracle responds with

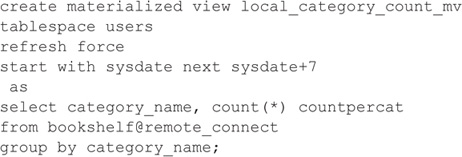
Image

Image

The command shown in the preceding example creates a read-only materialized view called LOCAL\_BOOKSHELF\_MV. Its underlying table is created in a tablespace named USERS. You can place materialized view logs in tablespaces apart from the materialized views they support. The FORCE refresh option is specified because no materialized view log exists on the base table for the materialized view; Oracle tries to use a FAST refresh but will use a COMPLETE refresh until the materialized view log is created. The materialized view’s query specifies that the entire BOOKSHELF table, with no modifications, is to be copied to the local database. As soon as the LOCAL\_BOOKSHELF\_MV materialized view is created, its underlying table is populated with the BOOKSHELF data. Thereafter, the materialized view will be refreshed every seven days. The storage parameters that are not specified will use the default values for those parameters for the USERS tablespace.

The following example creates a materialized view named LOCAL\_CATEGORY\_COUNT\_MV in a local database, based on a remote table named BOOKSHELF in a database accessed via the REMOTE\_CONNECT database link:

Image



The query in the LOCAL\_CATEGORY\_COUNT\_MV materialized view counts the number of books in each category in the remote BOOKSHELF table.

Here are a few important points about the two examples shown in this section:

image The GROUP BY query used in the LOCAL\_CATEGORY\_COUNT\_MV materialized view could be performed in SQL\*Plus against the LOCAL\_BOOKSHELF materialized view. That is, the GROUP BY operation can be performed outside of the materialized view.

image Since LOCAL\_CATEGORY\_COUNT\_MV uses a GROUP BY clause, it is a complex materialized view and may only be able to use COMPLETE refreshes. LOCAL\_BOOKSHELF\_MV, as a simple materialized view, can use FAST refreshes.

The two materialized views shown in the preceding examples reference the same table. Because one of the materialized views is a simple materialized view that replicates all columns and all rows of the master table, the second materialized view may, at first, appear to be redundant. However, sometimes the second, complex materialized view is the more useful of the two.

How can this be? First, remember that these materialized views are being used to service the query needs of local users. If those users always perform GROUP BY operations in their queries, and their grouping columns are fixed, then LOCAL\_CATEGORY\_COUNT\_MV may be more useful to them. Second, if the transaction volume on the master BOOKSHELF table is very high, or the master BOOKSHELF table is very small, there may not be a significant difference in the refresh times of the FAST and COMPLETE refreshes. The most appropriate materialized view is the one that is most productive for your users.

##### ****Types of Materialized Views****

The materialized views shown in the previous examples illustrated two types of materialized views. In the first, the materialized view created a local copy of remote data, with no aggregations. In the second, an aggregation was performed. Both of the base queries could be extended to include joins of multiple tables. The key distinguishing factor is the use of aggregations in the second example. A third type of materialized view is a nested materialized view—a materialized view whose definition is based on another materialized view.

If a materialized view contains an aggregate, a FAST refresh is still possible if the SELECT list contains all of the GROUP BY columns. There must also be a COUNT(\*) or COUNT(COLUMN\_ NAME) on any aggregated columns.

If a materialized view contains only joins but no aggregates, a FAST refresh is available after any INSERT, UPDATE, or DELETE to the base tables. The ROWID columns from each table must be present in the SELECT list for the materialized view’s base query, and all of the referenced tables must have materialized view logs.

Because they send the incremental changes from the referenced tables to the materialized view, FAST refreshes usually represent the fastest way to update the data in your materialized views.

##### ****ROWID vs. Primary Key–Based Materialized Views****

You can base materialized views on the master table’s primary key values instead of basing them on the master table’s ROWIDs. You should decide between these options depending on several factors:

image **System stability**   If the master site is not stable, then you may need to perform database recoveries involving the master table. When you use Oracle’s Data Pump Export and Import utilities to perform recoveries, the ROWID values of rows will change. If the system requires frequent exports and imports, you should use primary key–based materialized views.

image **Size of materialized view log table**   Oracle allows you to store the changes to master tables in separate tables called materialized view logs (described later in this chapter). If the primary key consists of many columns, the materialized view log table for a primary key–based materialized view may be considerably larger than the materialized view log for a comparable ROWID-based materialized view.

image **Referential integrity**   To use primary key–based materialized views, you must have defined a primary key on the master table. If you cannot define a primary key on the master table, then you must use ROWID-based materialized views.

##### ****Using Prebuilt Tables****

When you create a materialized view, you can specify BUILD IMMEDIATE to populate the materialized view immediately or BUILD DEFERRED to populate the materialized view later (via a complete refresh). If you need to carefully manage the transactions that initially populate the materialized view, you can create a table that has the same structure as the materialized view and populate it. When the table is fully loaded and properly indexed, use the ON PREBUILT TABLE clause of the CREATE MATERIALIZED VIEW command. The table and the materialized view must have the same name, and the table must have the same columns and datatypes as the materialized view (you can SPECIFY WITH REDUCED PRECISION to accommodate differences in precision). The table can contain additional, unmanaged columns.

Once the table has been registered as a materialized view, you can maintain it via refreshes, and the optimizer can use it in query rewrite operations. For query rewrite to work properly on a prebuilt table, you must set the QUERY\_REWRITE\_INTEGRITY initialization parameter to STALE\_ TOLERATED or TRUSTED.

##### ****Indexing Materialized View Tables****

When you create a materialized view, Oracle creates a local base table containing the data that satisfies the base query. Because that data has been replicated with a goal in mind (usually to improve performance in the database or the network), it is important to follow through on that goal after the materialized view has been created. As with regular tables, use indexes on materialized view columns to improve performance. Index columns that are frequently used in the WHERE clauses of queries; if a set of columns is frequently accessed in queries, create a concatenated index on that set of columns. (See [Chapter 46](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch46.html#ch46) for more information on the Oracle Optimizer.)

Oracle does not automatically create indexes for complex materialized views on columns other than the primary key. You need to create these indexes manually. To create indexes on your local base table, use the CREATE INDEX command (see the Alphabetical Reference). Do not create any constraints on the materialized view’s local base table; Oracle will maintain the constraint-based relationships on the master tables.

Because no indexes are created on the columns that users are likely to query from the materialized view, you should create indexes on the materialized view’s local base table.

#### ****Using Materialized Views to Alter Query Execution Paths****

For a large database, a materialized view may offer several performance benefits. You can use materialized views to influence the optimizer to change the execution paths for queries. This feature, called query rewrite, enables the optimizer to use a materialized view in place of the table queried by the materialized view, even if the materialized view is not named in the query. For example, if you have a large SALES table, you may create a materialized view that sums the SALES data by region. If a user queries the SALES table for the sum of the SALES data for a region, Oracle can redirect that query to use your materialized view in place of the SALES table. As a result, you can reduce the number of accesses against your largest tables, improving system performance. Further, because the data in the materialized view is already grouped by region, any summarization is already complete by the time the query is issued.

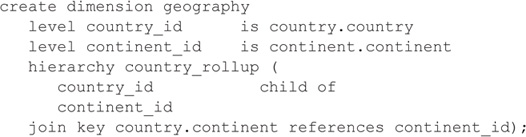
Image

**NOTE**

You must specify ENABLE QUERY REWRITE in the materialized view definition for the view to be used as part of a query rewrite operation.

To use the query rewrite capability effectively, you should create a dimension that defines the hierarchies within the table’s data. To execute the CREATE DIMENSION command, you need to have been granted the CREATE DIMENSION system privilege. For instance, you can create a dimension that supports the hierarchy between the COUNTRY and CONTINENT sample tables:

Image



To enable a materialized view for query rewrite, all of the master tables for the materialized view must be in the materialized view’s schema, and you must have the QUERY REWRITE system privilege. If the view and the tables are in separate schemas, you must have the GLOBAL QUERY REWRITE system privilege. In general, you should create materialized views in the same schema as the tables on which they are based; otherwise, you will need to manage the permissions and grants required to create and maintain the materialized view.

You can enable or disable query rewrite at the SQL statement level via the REWRITE and NOREWRITE hints. When using the REWRITE hint, you can specify materialized views for the optimizer to consider.

Image

**NOTE**

Query rewrite decisions are based on the costs of the different execution paths, so your statistics should be kept up to date.

For query rewrite to be possible, you must set the following initialization parameters:

image OPTIMIZER\_MODE = ALL\_ROWS or FIRST\_ROWS

image QUERY\_REWRITE\_ENABLED = TRUE

image QUERY\_REWRITE\_INTEGRITY = STALE\_TOLERATED, TRUSTED, or ENFORCED

By default, QUERY\_REWRITE\_INTEGRITY is set to ENFORCED; in this mode, all constraints must be validated. The optimizer only uses fresh data from the materialized views and only uses those relationships that are based on ENABLED VALIDATED primary, unique, or foreign key constraints. In TRUSTED mode, the optimizer trusts that the data in the materialized view is fresh and the relationships declared in dimensions and constraints are correct. In STALE\_TOLERATED mode, the optimizer uses materialized views that are valid but contain stale data, as well as those that contain fresh data.

If you set QUERY\_REWRITE\_ENABLED to FORCE, the optimizer rewrites queries to use materialized views even when the estimated query cost of the original query is lower.

If query rewrite occurs, the explain plan for the query (see [Chapter 46](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch46.html#ch46)) will list the materialized view as one of the objects accessed, along with an operation listed as “MAT\_VIEW REWRITE ACCESS.” You can use the DBMS\_MVIEW.EXPLAIN\_REWRITE procedure to see if rewrite is possible for a query, and which materialized views would be involved. If the query cannot be rewritten, the procedure will document the reasons.

EXPLAIN\_REWRITE takes three input parameters—the query, a materialized view name, and a statement identifier—and can store its output in a table. Oracle provides the CREATE TABLE command for the output table in a script named **utlxrw.sql** in the **./rdbms/admin** directory under the Oracle software home directory. The **utlxrw.sql** script creates a table named REWRITE\_TABLE.

You can query the REWRITE\_TABLE for the original cost, rewritten cost, and the optimizer’s decision. The MESSAGE column displays the reasons for the optimizer’s decision.

See the Data Warehousing Guide for further details and constraints related to query rewrite.

#### ****Using DBMS\_ADVISOR****

You can use the SQL Access Advisor to generate recommendations for the creation and indexing of materialized views. The SQL Access Advisor may recommend specific indexes (and types of indexes) to improve the performance of joins and other queries. The SQL Access Advisor may also generate recommendations for altering a materialized view so it supports query rewrite or FAST refreshes. You can execute the SQL Access Advisor from within Oracle Enterprise Manager or via executions of the DBMS\_ADVISOR package.

Image

**NOTE**

For best results from the DBMS\_ADVISOR package, you should gather statistics about all tables, indexes, and join columns prior to generating recommendations.

To use the SQL Access Advisor, either from Oracle Enterprise Manager or via DBMS\_ ADVISOR, follow these four steps:

**1**. Create a task.

**2**. Define the workload.

**3**. Generate recommendations.

**4**. View and implement recommendations.

You can create a task in one of two ways: by executing the DBMS\_ADVISOR.CREATE\_TASK procedure or by using the DBMS\_ADVISOR.QUICK\_TUNE procedure (as shown in the next section).

The workload consists of one or more SQL statements plus the statistics and attributes that relate to the statement. The workload may include all SQL statements for an application. The SQL Access Advisor ranks the entries in the workload according to statistics and business importance. The workload is created using the DBMS\_ADVISOR.CREATE\_SQLWKLD procedure. To associate a workload with a parent SQL Access Advisor task, use the DBMS\_ADVISOR.ADD\_SQLWKLD\_ REF procedure. If a workload is not provided, the SQL Access Advisor can generate and use a hypothetical workload based on the dimensions defined in your schema.

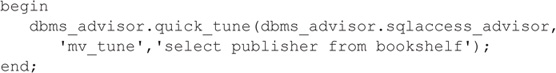
Once a task exists and a workload is associated with the task, you can generate recommendations via the DBMS\_ADVISOR.EXECUTE\_TASK procedure. The SQL Access Advisor considers the workload and the system statistics and attempts to generate recommendations for tuning the application. You can see the recommendations by executing the DBMS\_ADVISOR. GET\_TASK\_SCRIPT function or via data dictionary views. Each recommendation can be viewed via USER\_ADVISOR\_RECOMMENDATIONS (there are “ALL” and “DBA” versions of this view available, as well). To relate recommendations to a SQL statement, you need to use the USER\_ ADVISOR\_SQLA\_WK\_STMTS view and USER\_ADVISOR\_ACTIONS.

When you execute the GET\_TASK\_SCRIPT procedure, Oracle generates an executable SQL file that contains the commands needed to create, alter, or drop the recommended objects. You should review the generated script prior to executing it, particularly noting the tablespace specifications. In the following section, you will see how to use the QUICK\_TUNE procedure to simplify the tuning advisor process for a single command.

To tune a single SQL statement, use the QUICK\_TUNE procedure of the DBMS\_ADVISOR package. QUICK\_TUNE has two input parameters—a task name and a SQL statement. Using QUICK\_TUNE shields the user from the steps involved in creating workloads and tasks via DBMS\_ADVISOR.

For example, the following procedure call evaluates a query:

Image



Image

**NOTE**

The user executing this command needs the ADVISOR system privilege.

You can view the recommendation generated by QUICK\_TUNE via USER\_ADVISOR\_ ACTIONS, but it is easier to read if you use the DBMS\_ADVISOR procedures to generate a script file. The recommendation is that a materialized view be created to support the query. Because only one SQL statement was provided, this recommendation is given in isolation and does not consider any other aspects of the database or application.

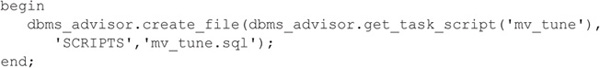
You can use the CREATE\_FILE procedure to automate the generation of a file containing the scripts needed to implement the recommendations. First, create a directory object to hold the file:

Image

Image

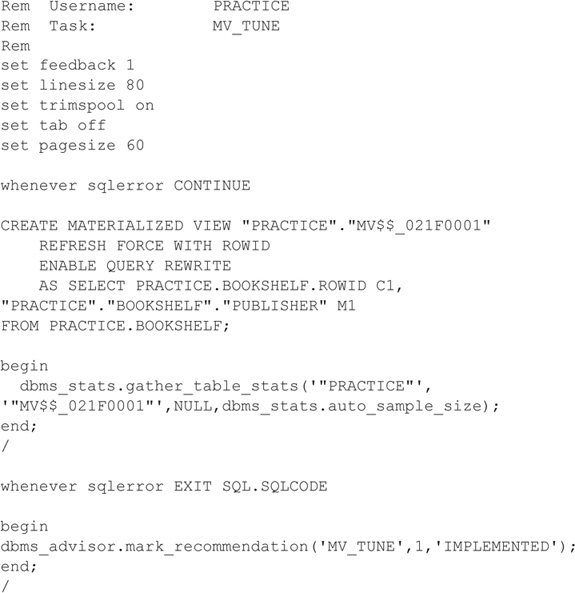
Next, execute the CREATE\_FILE procedure. It has three input variables: the script (generated via GET\_TASK\_SCRIPT, to which you pass the name of the task), the output directory, and the name of the file to be created:

Image



The **mv\_tune.sql** file created by the CREATE\_FILE procedure will contain commands similar to those shown in the following listing. Depending on the specific version of Oracle, the recommendations may differ.

Image



The MARK\_RECOMMENDATION procedure allows you to annotate the recommendation so it can be skipped during subsequent script generations. Valid actions for MARK\_ RECOMMENDATION include ACCEPT, IGNORE, IMPLEMENTED, and REJECT.

You can use the TUNE\_MVIEW procedure of the DBMS\_ADVISOR package to generate recommendations for reconfiguring your materialized views. TUNE\_MVIEW generates two sets of output results—for the creation of new materialized views and for the removal of previously created materialized views. The end result should be a set of materialized views that can be FAST refreshed, replacing materialized views that cannot be FAST refreshed.

You can view the TUNE\_MVIEW output via the USER\_TUNE\_MVIEW data dictionary view, or you can generate its scripts via the GET\_TASK\_SCRIPT and CREATE\_FILE procedures shown in the previous listings.

#### ****Refreshing Materialized Views****

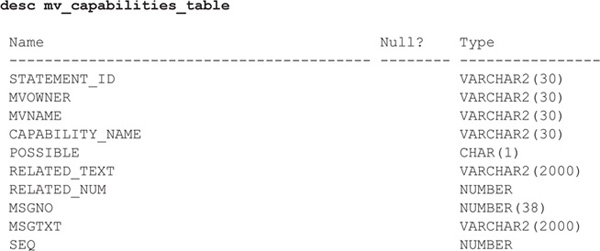
The data in a materialized view may be replicated either once (when the view is created) or at intervals. The CREATE MATERIALIZED VIEW command allows you to set the refresh interval, delegating the responsibility for scheduling and performing the refreshes to the database. In the following sections, you will see how to perform both manual and automatic refreshes.

##### ****What Kind of Refreshes Can Be Performed?****

To see what kind of refresh and rewrite capabilities are possible for your materialized views, you can query the MV\_CAPABILITIES\_TABLE table. The capabilities may change between versions, so you should reevaluate your refresh capabilities following Oracle software upgrades. To create this table, execute the **utlxmv.sql** script located in the **./rdbms/admin** directory under the Oracle software home directory.

The columns of MV\_CAPABILITIES\_TABLE are

Image



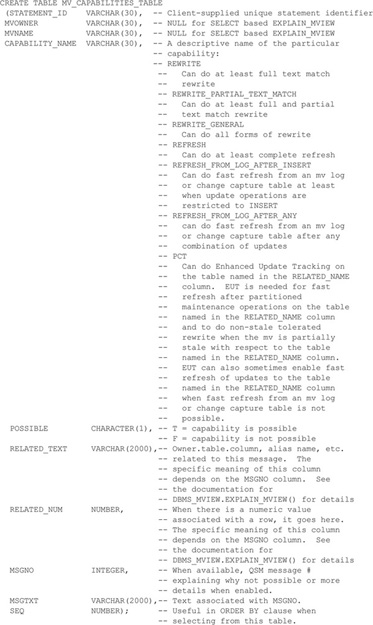
To populate the MV\_CAPABILITIES\_TABLE table, execute the DBMS\_MVIEW.EXPLAIN\_ MVIEW procedure, using the name of the materialized view as the input value as shown in the following listing:

Image

Image

The **utlxmv.sql** script provides guidance on interpreting the column values, as shown in the following listing:

Image



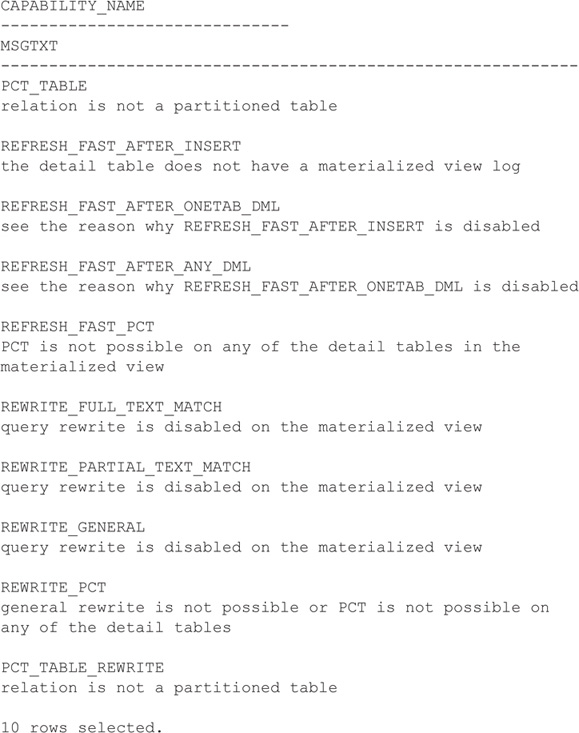
Once the EXPLAIN\_MVIEW procedure has been executed, you can query MV\_CAPABILITIES\_ TABLE to determine your options.

Image

Image

For the LOCAL\_BOOKSHELF\_MV materialized view, the query returns the following:

Image



Because the QUERY REWRITE clause was not specified when creating the materialized view, the query rewrite capabilities are disabled for LOCAL\_BOOKSHELF\_MV. FAST refresh capabilities are not supported because the base table does not have a materialized view log. If you change your materialized view or its base table, you should regenerate the data in MV\_CAPABILITIES\_ TABLE to see the new capabilities.

As shown in the preceding listing, the LOCAL\_BOOKSHELF\_MV materialized view cannot use a FAST refresh because its base table does not have a materialized view log. There are other constraints that will limit your ability to use FAST refreshes:

image The materialized view must not contain references to nonrepeating expressions such as SYSDATE and ROWNUM.

image The materialized view must not contain references to RAW or LONG RAW datatypes.

image For materialized views based on joins, ROWIDs from all tables in the FROM list must be part of the SELECT list.

image If there are outer joins, all the joins must be connected by ANDs, the WHERE clause must have no selections, and unique constraints must exist on the join columns of the inner join table.

image For materialized views based on aggregates, the materialized view logs must contain all columns from the referenced tables, must specify the ROWID and INCLUDING NEW VALUES clauses, and must specify the SEQUENCE clause.

See the Data Warehousing Guide for additional restrictions related to FAST refreshes of complex aggregates.

Image

**NOTE**

You can specify an ORDER BY clause in the CREATE MATERIALIZED VIEW command. The ORDER BY clause only affects the initial creation of the materialized view; it will not affect any refresh.

##### ****FAST Refresh with CONSIDER FRESH****

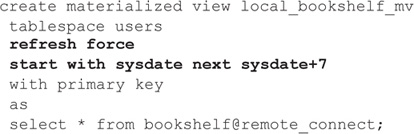
You may need to accumulate historical information in a materialized view, even though the detailed information is no longer in the base tables. As of Oracle 11g, you can use the CONSIDER FRESH clause of the ALTER MATERIALIZED VIEW command to instruct the database that the contents of the materialized view correctly reflect the base tables even if that is not the case.

For example, you may have a materialized view that summarizes data from a detail transactional table. If you remove historical data from the transactional table, the materialized view should be refreshed, because Oracle will know its data is stale. You can use the CONSIDER FRESH clause to change the status of the materialized view, enabling the materialized view to be used for query rewrites. You can then continue to take advantage of query rewrite capabilities while scheduling a refresh of the materialized view so it will accurately reflect the underlying data.

##### ****Automatic Refreshes****

Consider the LOCAL\_BOOKSHELF\_MV materialized view described earlier. Its refresh schedule settings, defined by its CREATE MATERIALIZED VIEW command, are shown in bold in the following listing:

Image



The refresh schedule has three components. First, the type of refresh (FAST, COMPLETE, NEVER, or FORCE) is specified. FAST refreshes use materialized view logs (described later in this chapter) to send changed rows from the master table to the materialized view. COMPLETE refreshes delete all rows from the materialized view and repopulate it. The FORCE option for refreshes tells Oracle to use a FAST refresh if it is available; otherwise, use a COMPLETE refresh.

The START WITH clause tells the database when to perform the first replication from the master table to the local base table. It must evaluate to a future point in time. If you do not specify a START WITH time but specify a NEXT value, Oracle will use the NEXT clause to determine the start time. To maintain control over your replication schedule, you should specify a value for the START WITH clause.

The NEXT clause tells Oracle how long to wait between refreshes. Because this clause’s value is applied to a different base time each time the materialized view is refreshed, the NEXT clause specifies a date expression instead of a fixed date. In the preceding example, the expression is

Image

Image

Every time the materialized view is refreshed, the next refresh is scheduled for seven days later. Although the refresh schedule in this example is fairly simple, you can use many of Oracle’s date functions to customize a refresh schedule. For example, if you want to refresh every Monday at noon, regardless of the current date, you can set the NEXT clause to

Image

Image

This example finds the next Monday after the current system date; the time portion of that date is truncated, and 12 hours is added to the date. (For information on date functions in Oracle, see [Chapter 10](https://www.safaribooksonline.com/library/view/oracle-database-12c/9780071801751/ch10.html#ch10).)

For automatic materialized view refreshes to occur, you must have at least one background snapshot refresh process running in your database. The refresh process periodically “wakes up” and checks whether any materialized views in the database need to be refreshed. The number of processes running in your database is determined by an initialization parameter called JOB\_ QUEUE\_PROCESSES. That parameter must be set (in your initialization parameter file) to a value greater than 0; for most cases, a value of 1 should be sufficient. A coordinator process starts job queue processes as needed.

If the database is not running the job queue processes, you need to use manual refresh methods, described in the next section.

##### ****Manual Refreshes****

In addition to the database’s automatic refreshes, you can perform manual refreshes of materialized views. These override the normally scheduled refreshes; the new START WITH value will be based on the time of your manual refresh.

To refresh a single materialized view, use DBMS\_MVIEW.REFRESH. Its two main parameters are the name of the materialized view to be refreshed and the method to use. For this method, you can specify “c” for a COMPLETE refresh, “f” for FAST refresh, “p” for a FAST refresh using Partition Change Tracking (PCT), and “?” for FORCE. For example:

Image

Image

Image

**NOTE**

Partition Change Tracking (PCT) occurs when partition maintenance operations have been performed on the tables referenced by the materialized view. In PCT, Oracle performs the refresh by recomputing the rows in the materialized view affected by the changed partitions in the detail tables, avoiding the need for a complete refresh.

If you are refreshing multiple materialized views via a single execution of DBMS\_MVIEW. REFRESH, list the names of all the materialized views in the first parameter, and their matching refresh methods in the second parameter, as shown here:

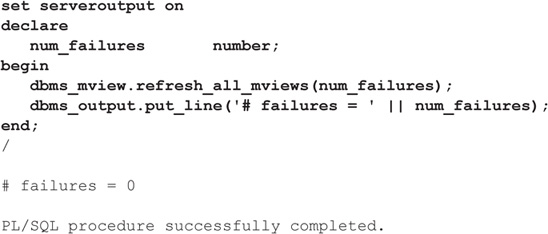
Image

Image

In this example, the materialized view named LOCAL\_BOOKSHELF\_MV will be refreshed via a FORCE refresh, whereas the second materialized view will use a COMPLETE refresh.

You can use a separate procedure in the DBMS\_MVIEW package to refresh all of the materialized views that are scheduled to be automatically refreshed. This procedure, named REFRESH\_ALL\_MVIEWS, refreshes each materialized view separately. It does not accept any parameters other than an output variable that returns the number of materialized view refresh failures. The following listing shows an example of its execution:

Image



Because the materialized views are refreshed via REFRESH\_ALL consecutively, they are not all refreshed at the same time. Therefore, a database or server failure during the execution of this procedure may cause the local materialized views to be out of sync with each other. If that happens, simply rerun this procedure after the database has been recovered. As an alternative, you can create refresh groups, as described in the next section.

REFRESH\_ALL\_MVIEWS refreshes all materialized views that have the following properties:

image The materialized view has not been refreshed since the most recent change to a master table or master materialized view on which the materialized view depends.

image The materialized view and all of the master tables or master materialized views on which the materialized view depends are local.

image The materialized view is in the view DBA\_MVIEWS.

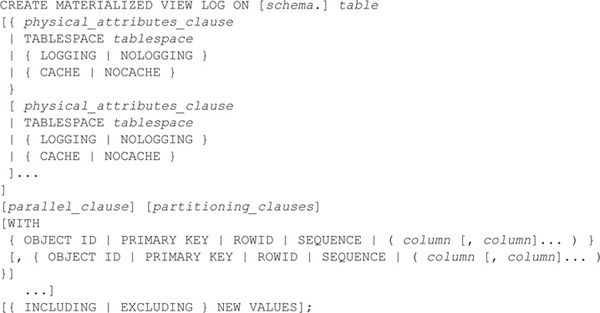
If you create nested materialized views, you can use the DBMS\_MVIEW.REFRESH\_ DEPENDENT procedure to ensure all materialized views within a tree are refreshed.

#### ****CREATE MATERIALIZED VIEW Log Syntax****

A materialized view log is a table that records changes to the rows in the master table and the materialized views’ replication history. The record of changed rows can then be used during refreshes to send to the materialized views only those rows that have changed in the master table. Multiple materialized views based on the same table can use the same materialized view log.

The full syntax for the CREATE MATERIALIZED VIEW LOG command is shown in the Alphabetical Reference. The following listing shows part of the syntax; as you may note from its syntax, it has all of the parameters normally associated with tables:

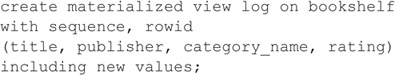
Image



The CREATE MATERIALIZED VIEW LOG command is executed in the master table’s database, usually by the master table’s owner. Do not create materialized view logs for tables that are only involved in complex materialized views (since they wouldn’t be used anyway). No name is specified for the materialized view log.

You can create a materialized view log for the BOOKSHELF table via the following command, executed from within the account that owns the table:

Image



The WITH SEQUENCE clause is needed to support the replication of mixed-DML operations against multiple base tables.

Because materialized view logs may grow unpredictably over time in production databases, you should consider storing their associated objects in tablespaces that are dedicated to materialized view logs.

Image

**NOTE**

As of Oracle 11g, the capture of changes in materialized view logs can be disabled for an individual session while logging continues for changes made by other sessions.

To create the materialized view log, you must have CREATE TABLE and CREATE TRIGGER system privileges. If you are creating the materialized view log from a user account that does not own the master table, you need to have CREATE ANY TABLE, COMMENT ANY TABLE, and CREATE ANY TRIGGER system privileges as well as the SELECT privilege on the materialized view master table.

#### ****Altering Materialized Views and Logs****

You may alter the storage parameters, refresh option, and refresh schedule for existing materialized views. If you are unsure of the current settings for a snapshot, check the USER\_ MVIEWS data dictionary view.

The syntax for the ALTER MATERIALIZED VIEW command is shown in the Alphabetical Reference. The command in the following listing alters the refresh option used by the LOCAL\_ BOOKSHELF\_MV materialized view:

Image

Image

All future refreshes of LOCAL\_BOOKSHELF\_MV will refresh the entire local base table.

To alter a materialized view, you must

image Own the materialized view, or

image Have the ALTER ANY MATERIALIZED VIEW system privilege

In contrast, to alter a materialized view log, you must

image Own the table, or

image Have the ALTER privilege on the table, or

image Have the ALTER ANY TABLE system privilege

If you create the materialized view log without the ROWID or the SEQUENCE clauses, you can add them after the fact via the ALTER MATERIALIZED VIEW command.

#### ****Dropping Materialized Views and Logs****

To drop a materialized view, you must have the system privileges required to drop both the materialized view and all of its related objects. You need to have the DROP MATERIALIZED VIEW system privilege if the object is in your schema or the DROP ANY MATERIALIZED VIEW system privilege if the materialized view is not in your schema.

The following command drops the LOCAL\_CATEGORY\_COUNT\_MV materialized view created earlier in this chapter:

Image

Image

Image

**NOTE**

When you drop a materialized view that was created on a prebuilt table, the table still exists but the materialized view is dropped.

You can drop materialized view logs via the DROP MATERIALIZED VIEW LOG command. Once the materialized view log is dropped from a master table, FAST refreshes cannot be performed for simple materialized views based on that table. Therefore, you should drop a materialized view log when no simple materialized views are based on the master table. The following command drops the materialized view log that was created on the BOOKSHELF table earlier in this chapter:

Image

Image

To drop a materialized view log, you must have the ability to drop both the materialized view log and its related objects. If you own the materialized view log, you must have the DROP TABLE and DROP TRIGGER system privileges. If you do not own the materialized view log, you need the DROP ANY TABLE and DROP ANY TRIGGER system privileges to execute this command.