

CHAPTER 13 Indexes, Sequences, and Views

Oracle SQL By Example, Fourth Edition by Alice Rischert. Published by Prentice Hall. Copyright © 2008 by Pearson Education, Inc.

CHAPTER OBJECTIVES

In this chapter, you will learn about:

- ▶ Indexes
- ▶ Sequences
- ▶ Views

This chapter covers three different yet very important database objects: indexes, sequences, and views. Indexes are required for good performance of any database. A well-thought-out indexing strategy entails the careful placement of indexes on relevant columns. In this chapter, you will gain an understanding about the advantages and trade-offs when using indexes on tables.

Sequences generate unique values and are used mainly for creating primary key values. In this chapter, you will learn how to create and use sequences.

Views are significant in a database because they can provide row-level and column-level security to the data; they allow you to look at the data differently and/or display only specific information to the user. Views are also useful for simplifying the writing of queries for end users because they can hide the complexities of joins and conditional statements.

571

572

LAB 13.1 Indexes

LAB OBJECTIVES

After this lab, you will be able to:

- ▶ Create Indexes
- ▶ Understand the Impact of Indexes

To achieve good performance for data retrieval and data manipulation statements, you need to understand Oracle's use of indexes. Just as you use the index in the back of a book to quickly find information, Oracle uses indexes to speed up data retrieval. If the appropriate index does not exist on a table, Oracle needs to examine every row. This is called a *full table scan*.

If an index decreases query time, why not just index every column in a table? When you retrieve a large number of rows in a table, it might be more efficient to read the entire table rather than look up the values from the index. It also takes a significant amount of time and storage space to build and maintain an index. For each DML statement that changes a value in an indexed column, the index needs to be maintained.

The B-Tree Index

In this book, you will perform exercises related to Oracle's most popular index storage structure—the B-tree index. The merits and uses of another type of index, the bitmap index, are discussed briefly at the end of the lab; this type of index can be created only in Oracle's Enterprise Server Edition.

The B-tree (balanced tree) index is by far the most common type of index. It provides excellent performance in circumstances in which there are many distinct values on a column or columns. If you have several low-selectivity columns, you can also consider combining them into one *composite index*, also called a *concatenated index*. B-tree indexes are best for exact

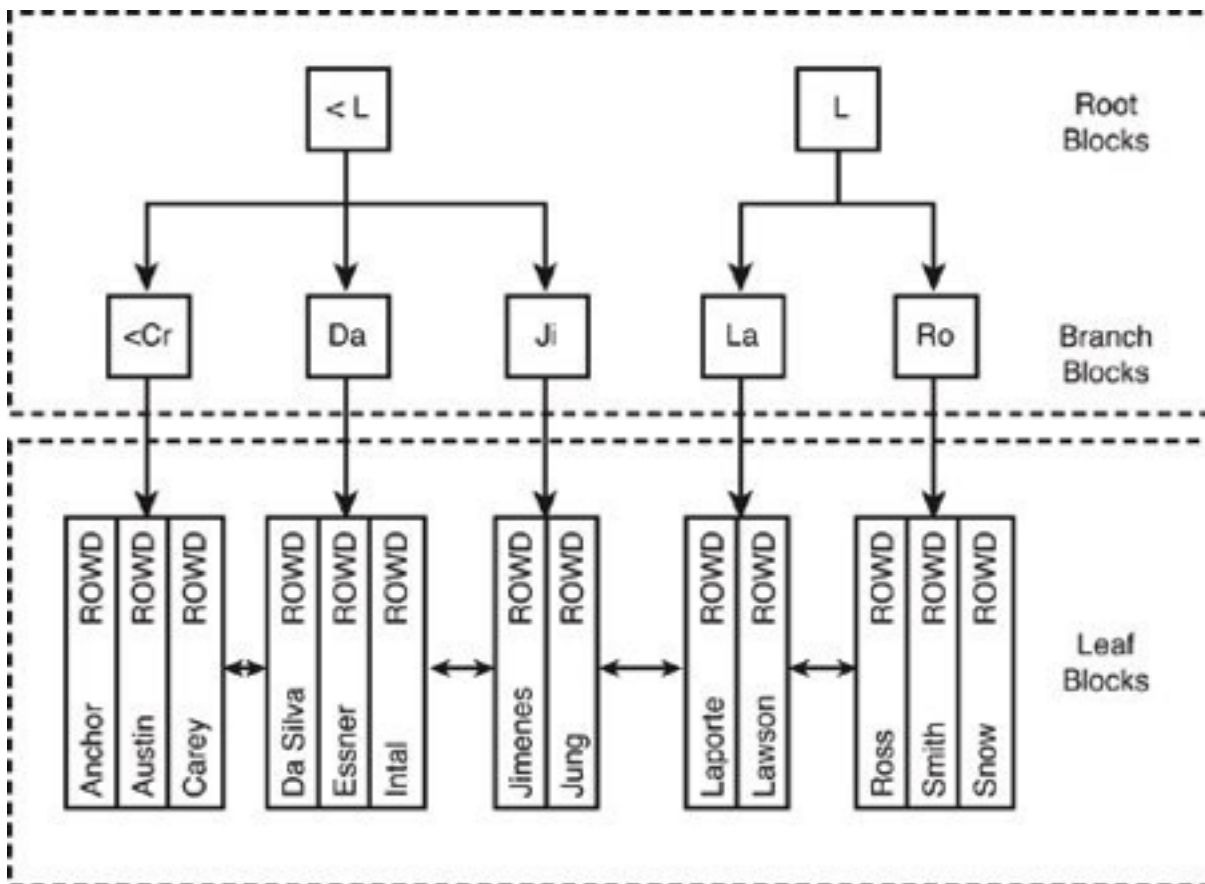
match and range searches against both small and very large tables.

[Figure 13.1](#) shows the structure of a B-tree index. It looks like an inverted tree and consists of two types of blocks: *root/branch blocks* and *leaf blocks*. Root or branch blocks are used for storing the key together with a pointer to the child block containing the key; leaf blocks store the key values along with the ROWID, which is the physical storage location for the data in the table.

572

573

FIGURE 13.1 B-tree index



SEARCHING FOR VALUES IN A B-TREE INDEX

The first step in searching for values in a B-tree index is to start with the root block of the index. The searched value is compared with the root block keys. For example, if you are looking for a student with the last name Essner, you must go down the root block < L. This block points to the next leaf blocks, which are greater than Da and less than Ji; going down on this leaf block, you find the value Essner and the associated ROWID, the physical address of the row. A leaf block also contains links to the next and previous leaf blocks, which allow scanning the index for ranges.

THE ROWID PSEUDOCOLUMN

A pseudocolumn is not an actual column, but it acts like one. One of the pseudocolumns you have already used is ROWNUM, which restricts the number of rows a query returns. The ROWID pseudocolumn is a unique address to a particular row, and every row has a ROWID.

Indexes store the ROWID to retrieve rows quickly because the ROWID consists of several components: the data object number, the number of the data block,

the number of rows within the data block, and the data file number. The data block and the data file define the physical storage characteristics of data within the individual Oracle database. This allows you to quickly access the physical row. Following is an example of a ROWID value.

```
SELECT ROWID, student_id, last_name
FROM student
```

```
WHERE student_id = 123

ROWID                STUDENT_ID LAST_NAME
-----
AAADA1AABAAAR1AAD      123 Radicola

1 row selected.
```

573

574



A ROWID is always unique. It is the fastest

way to access a row.

Rather than make Oracle search through the index, you can use the ROWID in UPDATE statements to directly access the row. For example, because the ROWID of the student named Radicola is already selected as part of the query, a subsequent update to the name of the student can find the row in the table immediately, without having to scan the entire table or use an index.

```
UPDATE student
    SET last_name =
        'Radicolament'
    WHERE ROWID =
        'AAADA1AABAAARAIAAD'
    AND last_name = 'Radicola'
    AND student_id = 123
```

You cannot update the ROWID, but the ROWID may change if you first delete the row and then reinsert the row because it can now be placed in another physical location. If your table moves to another database instance or another schema, the ROWID will be different.

As you learned in [Chapter 11](#), “Insert, Update, and Delete,” it is always good practice to include the old values in the WHERE clause of UPDATE to ensure that another session or user has not changed the name in the meantime.

Creating an Index

You create an index by using the following general syntax.

```
CREATE [UNIQUE|BITMAP] INDEX
    indexname
```

```
ON tablename
(column|col_expression [ASC|DESC]
[,column|col_expression [ASC|
DESC]]...)
[PCTFREE integer]
[TABLESPACE tablespacename|
DEFAULT]
[STORAGE ([INITIAL integer [K|M]]
[NEXT integer [K|M]])]
[ONLINE]
[VISIBLE|INVISIBLE]
[LOGGING|NOLOGGING]
[NOPARALLEL|PARALLEL [INTEGER]]
```

574

575

The following statement creates an index named SECT_LOCATION_I on the LOCATION column of the SECTION table.

```
CREATE INDEX sect_location_i
ON section(location)
```

Index created.

A subsequent query to find all the classes held in LOCATION L206 can take advantage of this index. Oracle looks up the value in the index. This retrieves the row faster than reading every row, particularly if the table has many records.


```
SELECT course_no, section_no, start_date_time, location
FROM section
WHERE location = 'L206'
COURSE_NO SECTION_NO START_DAT LOCATION
-----
120          2 24-JUL-07 L206

1 row selected.
```

Composite Indexes

Sometimes, it is useful to build indexes based on multiple columns; this type of index is called a *composite index*, or *concatenated index*. For example, you can create a composite index on two columns with a low selectivity (that is, not many distinct values). The combination of these low-selectivity values makes the composite index more selective. When you compare the query access time of a composite index to that of two individual single-column indexes, you find that the composite index offers better performance.

The following statement creates a composite index on the columns DESCRIPTION and COST. The first column of the index, also called the *leading edge* of the index, is the DESCRIPTION column; the second column of the index is the COST column.

```
CREATE INDEX  
course_description_cost_i  
ON course (description, cost)
```

Columns that are used together frequently in a WHERE clause and combined with the AND logical operator are often good candidates for a composite index, particularly if their combined selectivity is high. The order of the individual columns in the index can affect query performance. Choose the column you use most frequently in the WHERE clause first. If both columns are accessed with equal frequency, then choose the column with the highest selectivity. In this example, the COST column has very few distinct values and is therefore considered a low-selectivity column; access against an index with a low-selectivity column as the leading edge requires more index block reads and is therefore less desirable.

There are some caveats about composite indexes you must know about when writing queries. When executed in Oracle versions prior to 9i, a query such as the following, using the COST column in the WHERE clause, cannot use the COURSE_DESCRIPTION_COST_I index because it is not the leading edge of the index.

```
SELECT course_no, description, cost
FROM course
WHERE cost = 1095
```

However, Oracle can use a technique called *skip scan*, which may use the index nonetheless. As you work your way through this lab, you will learn more about this feature.

To find out what columns of a table are indexed and the order of the columns in an index, you can query the data dictionary views `USER_INDEXES` and `USER_IND_COLUMNS` or review the Indexes tab in SQL Developer for the respective table.

NULLs and Indexes

NULL values are not stored in a B-tree index, unless it is a composite index where at least the first column of the index contains a value. The following query does not make use of the single-column index on the `FIRST_NAME` column.

```
SELECT student_id, first_name
FROM student
WHERE first_name IS NULL
```

Functions and Indexes

Even when you create an index on one or multiple columns of a table, Oracle may not be able to use it. In the next scenario, assume that the `LAST_NAME` column of the `STUDENT` table is indexed. The following SQL query applies the `UPPER` function on the `LAST_NAME` column. You can use this `WHERE` clause expression if you don't know how the last name is stored in the column—it may be stored with the first initial in uppercase, in all uppercase, or perhaps in mixed case. This query does not take advantage of the index because the column is modified by a function.

```
SELECT student_id, last_name,  
       first_name  
       FROM student  
       WHERE UPPER(last_name) = 'SMITH'
```

You can avoid this behavior by creating a function-based index instead, as in the following example. This allows for case-insensitive searches on the `LAST_NAME` column.

```
CREATE INDEX stud_last_name_1  
       ON student(UPPER(last_name))
```

Indexes and Tablespaces

To optimize performance, it is important that you separate indexes from data by placing them in separate tablespaces, residing on different physical devices. This significantly improves the performance of your queries. Use the following statement to create an index named SECT_LOCATION_I on a tablespace called INDEX_TX, with an initial size of 500K and 100K in size for each subsequent extent.

```
CREATE INDEX sect_location_i
ON section(location)
TABLESPACE index_tx
STORAGE (INITIAL 500K NEXT 100K)
```

576

577

The storage clause of indexes is similar to the storage clause discussed in [Chapter 12](#), “Create, Alter, and Drop Tables”; however, the PCTUSED parameter is not applicable for indexes. If you want to see a list of tablespaces accessible to you, query the data dictionary view USER_TABLE-SPACES.

Unique Index Versus Unique Constraint

At times you might want to enforce a unique combination of the values in a table (for example, the COURSE_NO and SECTION_NO columns of the SECTION table). You can create a unique constraint on the table that automatically creates a unique index.

```
ALTER TABLE section
  ADD CONSTRAINT sect_sect2_uk
  UNIQUE
    (section_no, course_no)
  USING INDEX
  TABLESPACE index_tx
  storage (initial 12K next 12K)
```

Or you can use the CREATE UNIQUE INDEX command.

```
CREATE UNIQUE INDEX
  section_sect_course_no_i
  ON section (section_no,
  course_no)
  TABLESPACE index_tx
  storage (initial 12K next 12K)
```

Oracle prefers that you use the unique constraint syntax for future compatibility.

Indexes and Constraints

When you create a primary key constraint or a unique constraint, Oracle creates the index automatically unless a suitable index already exists. In [Chapter 12](#), you learned about various syntax options you can use.

The index NEW_TERM_PK is created as part of the CREATE TABLE statement and is associated with the primary key constraint.

```
CREATE TABLE new_term
    (term_no NUMBER(8) NOT NULL
    PRIMARY KEY USING INDEX
    (CREATE INDEX new_term_pk ON
    new_term(term_no)
    STORAGE (INITIAL 100 K NEXT
    100K)) ,
    season_tx VARCHAR2(20) ,
    sequence_no NUMBER(3))
```

577

578

The advantage of using this syntax is that you can create an index in the same statement of the CREATE TABLE command, whereby you have control over the storage characteristics of the index. It doesn't require two

separate statements: a CREATE TABLE statement and an ALTER TABLE statement that adds the constraint and the index plus storage clause.

If you already have an existing index and you want to associate a constraint with it, you can use a statement similar to the following. It assumes an existing index called SEMESTER_SEMESTER_ID_I, based on the SEMESTER_ID column.

```
ALTER TABLE semester
  ADD CONSTRAINT semester_pk
  PRIMARY KEY (semester_id)
  USING INDEX
  semester_semester_id_i
```

The next statement shows an example of a unique constraint that is associated with a unique index.

```
CREATE TABLE semester
  (semester_id NUMBER(8),
   semester_name VARCHAR2(8) NOT
  NULL,
   year_no NUMBER(4) NOT NULL,
   CONSTRAINT semester_uk UNIQUE
  (semester_name, year_no)
  USING INDEX
  (CREATE UNIQUE INDEX
  semester_sem_yr_uk
```



```
ON semester(semester_name,  
year_no) ) )
```



When disabling a unique or primary key, you

can keep the index if you specify the **KEEP INDEX** clause in an **ALTER TABLE** statement (see [Chapter 12](#)).

Indexes and Foreign Keys

You should almost always index foreign keys because they are frequently used in joins. In addition, if you intend to delete or update unique or primary keys on the parent table, you should index the foreign keys to improve the locking of child records. Foreign keys that are not indexed require locks to be placed on the entire child table when a parent row is deleted or the primary or unique keys of the parent table are updated. This prevents any insertions, updates, and deletions on the entire child table until the row is committed or rolled back. The advantage of an index on the foreign key column is that the locks are placed on the affected indexed child rows, thus not locking up the entire child table. This is more efficient and allows data manipulation of child rows *not* affected by the updates and deletions of the parent table.

This key issue has caused headaches for many unwitting programmers who spent days reviewing their code for performance improvements. The lack of a foreign index key frequently turns out to be the culprit for the slow performance of updates and deletions.

578

579

Dropping an Index

To drop an index, use the DROP INDEX command. You might drop an index if queries in your applications do not utilize the index. You find out which indexes are used by querying the V\$OBJECT_USAGE data dictionary view.

```
DROP INDEX sect_location_i  
Index dropped.
```



When you drop a table, all associated indexes are dropped automatically.

Bitmap Indexes

Oracle supports bitmap indexes, which are typically used in a data warehouse where the primary goal is querying and analyzing data, with bulk data loads occurring at certain intervals. Bitmap indexes are not suitable for tables with heavy data manipulation activity by many

users because any such changes on this type of index may significantly slow down the transactions. A bitmap index is typically used on columns with a very low selectivity—that is, columns with very few distinct values. For example, a column such as GENDER—with the four distinct values female, male, unknown, and not applicable (in case of a legal entity such as a corporation)—has a very low selectivity.

A low selectivity is expressed as the number of distinct values as a total against all the rows in the database. For example, if you had 9,000 distinct values in a table with one million rows, it would be considered a low-selectivity column. In this scenario, the number of distinct values represents less than 1 percent of the entire rows in the table, and this column may be a viable column choice for a bitmap index.

[Figure 13.2](#) illustrates the concept of a bitmap index. The example shows a hypothetical CUSTOMER table with a bitmap index on the GENDER column. The bitmap index translates the distinct values for the GENDER column of individual customers. In this simplified example, the customers with the IDs 1 and 2 have GENDER = F, which makes the bit turned on to 1. The other values, such as GENDER = M, GENDER = N/A, and GENDER = UNKNOWN, have a 0, indicating that

these values are not true for the row. The next customer, with the ID of 3, has the 1 bit turned on GENDER = M, the other values are zero.

FIGURE 13.2 A bitmap index

CUSTOMER Table

ID	FIRST_NAME	LAST_NAME	GENDER
1	Mary	Jones	F
2	Carol	Smith	F
3	Fred	O1son	M
4		ABC, Inc.	N/A
...

Bitmapped index on GENDER column

CUSTOMER ID	1	2	3	4	...
GENDER = F	1	1	0	0	...
GENDER = M	0	0	1	0	...
GENDER = N/A	0	0	0	1	...
GENDER = Unknown	0	0	0	0	...

The following statement creates a bitmap index on the GENDER column of a CUSTOMER table.

```
CREATE BITMAP INDEX
customer_bm_gender_i
ON customer(gender)
```

If you have multiple bitmap indexes, such as one for GENDER, MARITAL STATUS, and ZIP, and you need to retrieve rows based on certain AND and OR conditions, then bitmap indexes perform very fast. They

quickly compare and merge the bit settings of these conditions and are therefore highly effective for large tables. Bitmap indexes require less storage space than traditional B-tree indexes, but they do not perform as well for less-than or greater-than comparisons. Bitmap indexes are available only with Oracle's Enterprise Edition.

Bitmap Join Indexes

The bitmap join index is another type of index that reduces the amount of data to be joined during a query. Essentially, it precomputes the join and stores the result in a bitmap; this type of index is useful in data warehousing environments. The following statement shows the creation of such an index.

```
CREATE BITMAP INDEX  
enroll_bmj_student_i  
ON enrollment(e.student_id)  
FROM enrollment e, student s  
WHERE e.student_id = s.student_id  
Index created.
```

Guidelines for When to Index

You want to consider indexing columns frequently used in the WHERE clause of SQL statements and foreign key

columns. Oracle automatically creates a unique index to enforce the primary key constraint and the unique constraint. The following are some general guidelines when an index is typically useful.

580

581

- ▶ Frequently accessed columns containing highly selective data for B-tree indexes.
- ▶ Columns frequently accessed with a small range of values for bitmap indexes.
- ▶ Columns that are frequently accessed and that contain many null values, but the query is looking for the NOT NULL values.
- ▶ Frequent queries against large tables retrieving less than 5 to 15 percent of the rows. The percentage may vary, depending on a number of factors, including the size of the table.

Building an index is often useless if:

- ▶ The table is small, but you should nevertheless create unique and primary constraints to enforce business rules.
- ▶ The query retrieves more than 5 to 15 percent of the rows.

- The indexed column is part of an expression. In this case, consider creating a function-based index instead.

In [Chapter 18](#), “SQL Optimization,” you’ll learn to verify that SQL statements issued actually use an index.

Although adding indexes may improve performance of certain queries, you must realize that Oracle may use this new index for other queries that previously used a different index. This rarely has an adverse effect, but you must nevertheless make certain that your overall application performance does not suffer because of this change.

Keep in mind that adding indexes may increase the time required for data manipulation operations, such as INSERT, UPDATE, and DELETE. If you primarily query the table, then creating the index may offset the disadvantage of additional time required for DML statements.

Altering an Index

A number of syntax options let you change various characteristics of an index, such as renaming or

rebuilding the index or altering the storage clause. The following are some of the general syntax options.

```
ALTER INDEX indexname
  [STORAGE ([NEXT integer [K|M] ] ) ]
  [REBUILD [ONLINE]
  [RENAME TO newindexname]
  [VISIBLE | INVISIBLE]
```

The following SQL statement shows the rebuild of an index. Periodically, you need to rebuild an index to compact the data and balance the index tree. This is particularly important after data is subject to a large number of DML changes; the rebuild operation will improve the performance of the queries. Using the index REBUILD option is faster than dropping and re-creating the index; furthermore, the index continues to be available for queries while the REBUILD operation is in progress.

581

582

```
ALTER INDEX stu_zip_fk_i REBUILD
Index altered.
```

Because a DDL command requires exclusive access to the table or index, other sessions issuing any DML commands prevent such changes. Therefore, data structure changes are usually performed during times when users are not accessing the system. However, you

can create or rebuild indexes with the ONLINE option while users are performing DML commands.

```
ALTER INDEX stu_zip_fk_i REBUILD  
ONLINE
```

Invisible Indexes

Oracle 11g added the ability to make an index invisible. The Oracle optimizer, which evaluates the best and most efficient access path for a SQL statement, ignores the index. This feature may come in handy if you consider removing a possibly unused index without dropping it to ensure that there is not an adverse impact on your application. If it turns out that the index is required after all, it is easy to make the index visible again without having to spend time and computing resources to re-create it.

Another possible use of this feature is when you want to use the index only for certain applications that find this index useful. These applications can execute an ALTER SESSION command and make the index visible.

```
ALTER SESSION  
SET  
OPTIMIZER_USE_INVISIBLE_INDEXES=TRUE
```

Although the index may be invisible to users, DML operations still have to maintain the invisible index. The index still remains; however, the index is no longer used by Oracle's optimizer.

To change the visibility on an index, use the following syntax.

```
ALTER INDEX indexname INVISIBLE
ALTER INDEX indexname VISIBLE
```

You can also create an index as invisible.

```
CREATE INDEX indexname
ON tablename(columnname) INVISIBLE
```

582

583

Loading Large Amounts of Data

When you insert or update large amounts of data, you might want to consider dropping certain indexes not used for the DML operation's WHERE clause to improve performance. After the operation is complete, you can re-create the appropriate indexes.

One fast way to re-create indexes is by using the NOLOGGING option. It avoids writing to the redo log, which keeps track of all the database changes. If you incur a fatal database error and you need to recover from

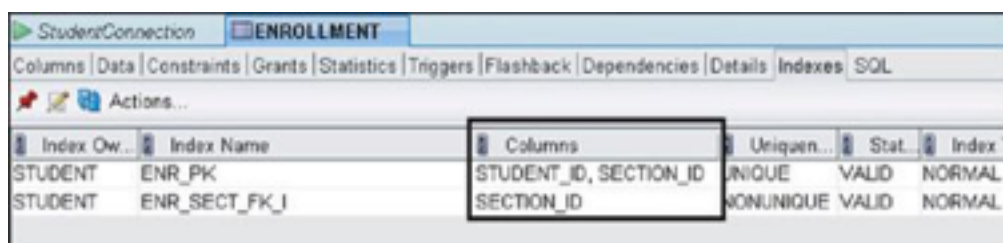
the redo log, the index will not be recovered. This may be fine because an index can always be re-created.

You can also create an index by using the **PARALLEL** option. This allows parallel scans of the table to create the index and can make index creation much faster, provided that you have the appropriate hardware configuration, such as multiple CPUs.

Indexes and SQL Developer

SQL Developer allows you to easily determine the existing indexes and their indexed columns on a given table. When you double-click one of the tables and review the **Indexes** tab, you see a list similar to the one shown in [Figure 13.3](#). This example shows a list of indexes for the **ENROLLMENT** table. The table displays the primary key index consisting of the two columns. The second index is on the **SECTION_ID** column, which is a foreign key column.

FIGURE 13.3 List of indexes for the ENROLLMENT table



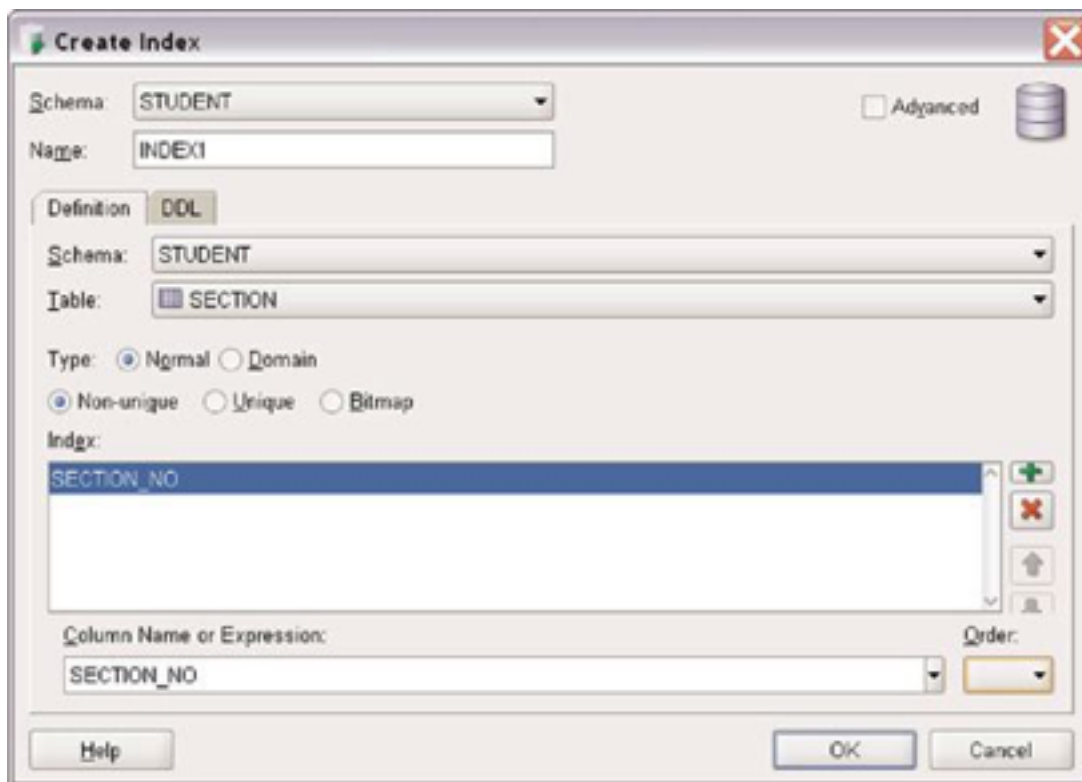
Index Ow...	Index Name	Columns	Uniquen...	Stat...	Index T
STUDENT	ENR_PK	STUDENT_ID, SECTION_ID	UNIQUE	VALID	NORMAL
STUDENT	ENR_SECT_FK_I	SECTION_ID	NONUNIQUE	VALID	NORMAL

Another way to access index information in SQL Developer is to use the Indexes node in the Connections pane. To create a new index you right-click on the Indexes node and select the New Index menu choice. This will bring up a screen similar to [Figure 13.4](#). You choose the indexed columns at the bottom of the screen. The Order drop-down box on the right of the column name or expression allows for ASC or DESC ordering of the columns within the index. This can be useful when you expect to retrieve the result columns in this order.

583

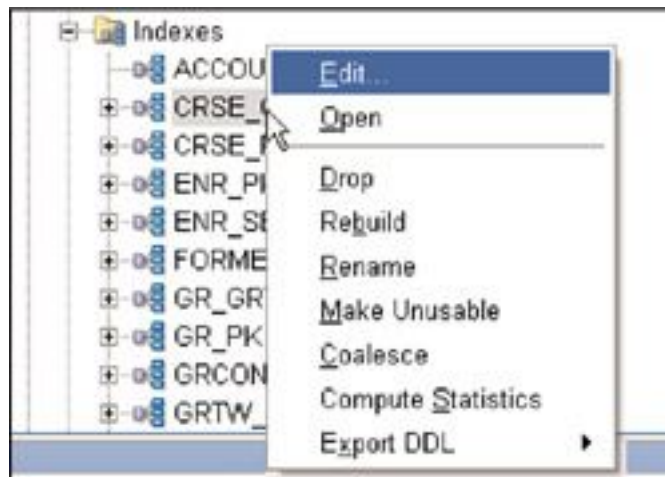
584

FIGURE 13.4 Create Index screen in SQL Developer



You can find a number of different index options by using SQL Developer's menu screens when you right-click on a specific index (see [Figure 13.5](#)).

FIGURE 13.5 Menu options for an index



[Table 13.1](#) highlights the functionality and purpose of each menu option.

584

585

TABLE 13.1 Overview of SQL Developer Index-Related Menu Choices

MENU CHOICE	DESCRIPTION
Edit	Allows you to modify properties related to the index, such as its columns or storage parameters. This results in a drop and re-creation of the index, as you can see when you click the DDL tab.
Open	Has the same effect as a double-click on the index name's node; this displays the column listing of the index, the data dictionary details, statistics, partition information, and the SQL DDL to re-create the index.
Drop	Drops the index.
Rebuild	Rebuilds the index. Creates a new index, based on the old index.
Rename	Renames the index.
Make Unusable	Makes the index unusable for SQL statements. To reuse the index, it must be either rebuilt or dropped and then re-created. Consider using the INVISIBLE option instead.
Coalesce	Merges index blocks to create a more compact and efficient index; useful when many deletes occurred against the table and its affected index.

Compute Statistics	Gathers the latest statistics (see Chapter 18).
Export DDL	Exports the index's DDL to a file, the Clipboard, or the SQL Worksheet window.

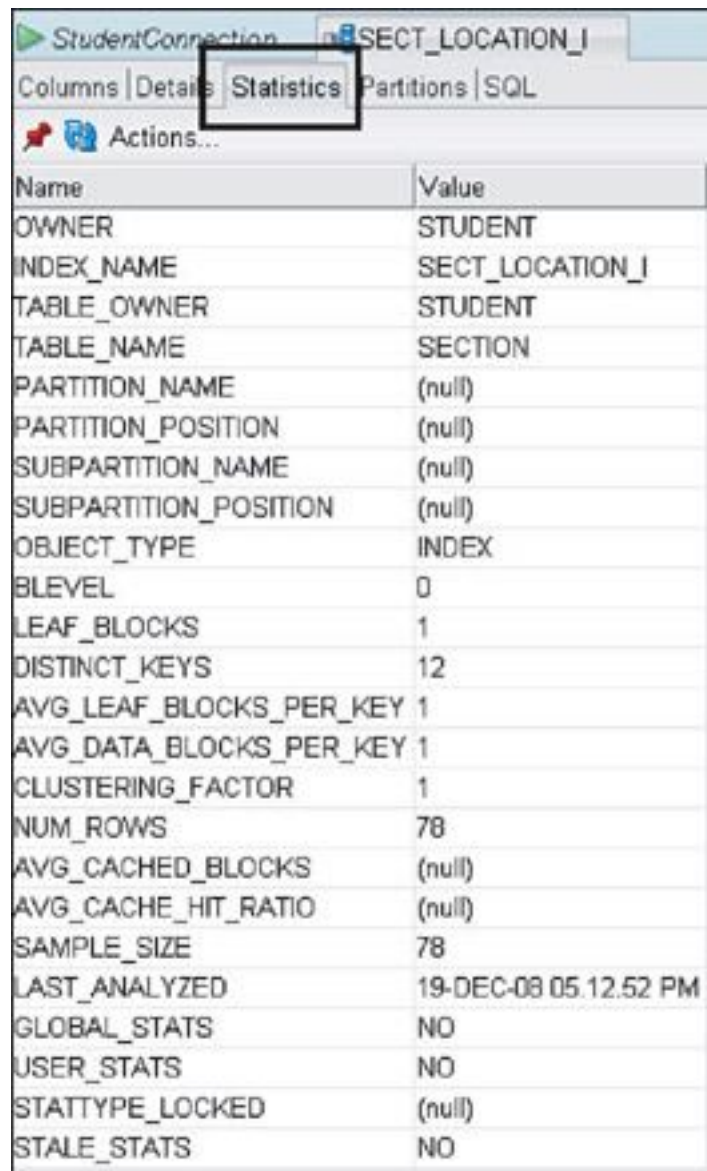
Indexes and Statistics

The Oracle optimizer makes a determination on how to execute your SQL statement. If an index exists, the optimizer evaluates whether the index is useful in achieving good performance. Because there are so many variables when determining how to best process a SQL statement, the Oracle optimizer relies on statistics about the tables and indexes, such as the number of rows and the number of distinct column values, among other variables.

These statistics need to be updated periodically to ensure good judgment by the optimizer. With Oracle version 10g and above, when you create an index, Oracle automatically collects the statistics. You can see the information in the Statistics tab of the given index within SQL Developer (see [Figure 13.6](#)).

585

FIGURE 13.6 The Statistics tab for the SECT_LOCATION_I index in SQL Developer



Name	Value
OWNER	STUDENT
INDEX_NAME	SECT_LOCATION_I
TABLE_OWNER	STUDENT
TABLE_NAME	SECTION
PARTITION_NAME	(null)
PARTITION_POSITION	(null)
SUBPARTITION_NAME	(null)
SUBPARTITION_POSITION	(null)
OBJECT_TYPE	INDEX
BLEVEL	0
LEAF_BLOCKS	1
DISTINCT_KEYS	12
AVG_LEAF_BLOCKS_PER_KEY	1
AVG_DATA_BLOCKS_PER_KEY	1
CLUSTERING_FACTOR	1
NUM_ROWS	78
AVG_CACHED_BLOCKS	(null)
AVG_CACHE_HIT_RATIO	(null)
SAMPLE_SIZE	78
LAST_ANALYZED	19-DEC-08 05:12:52 PM
GLOBAL_STATS	NO
USER_STATS	NO
STATTYPE_LOCKED	(null)
STALE_STATS	NO

The Index Details tab is much like the Details tab for a table. On it, you find information such as the storage

parameters, the date created, and the date the index was last analyzed; this tab primarily reflects data from the data dictionary view `ALL_INDEXES`.

The Partitions tab shows whether the index is split into multiple partitions that can be managed independently. Index and table partitioning are useful for large-scale databases.

LAB 13.1 EXERCISES

- a) Create an index on the `PHONE` column of the `STUDENT` table. Drop the index after you successfully create it to return the `STUDENT` schema to its original state.
- b) Create a composite index on the `FIRST_NAME` and `LAST_NAME` columns of the `STUDENT` table. Drop the index when you have finished.
- c) Create an index on the `DESCRIPTION` column of the `COURSE` table. Note that queries against the table often use the `UPPER` function. Drop the index after you successfully create it.

- d) Execute the following SQL statements. Explain the reason for the error.

```
CREATE TABLE test (col1 NUMBER)
```

586

587

```
CREATE INDEX test_coll_i ON  
test(coll)  
DROP TABLE test  
DROP INDEX test_coll_i
```

- e) Would a B-tree index work on a frequently accessed column with few distinct values? Explain.
- f) List the advantages and disadvantages of indexes on performance.
- g) Assume that an index exists on the column ENROLL_DATE in the ENROLLMENT table. Change the following query so it uses that index.

```
SELECT student_id, section_id,  
       TO_CHAR(enroll_date, 'DD-MON-  
YYYY')  
FROM enrollment  
WHERE TO_CHAR(enroll_date, 'DD-  
MON-YYYY') = '12-MAR-2007'
```

LAB 13.1 EXERCISE ANSWERS

- a) Create an index on the PHONE column of the STUDENT table. Drop the index after you successfully create it to return the STUDENT schema to its original state.

ANSWER: To create the index on the table, issue a **CREATE INDEX** statement.

```
CREATE INDEX stu_phone_i  
ON student(phone)
```

Index created.

Include the name of the table and the indexed column(s) in the index name; this convention allows easier identification of indexes and their respective columns in a particular table. As always, you can use SQL Developer to review the index listings or query the data dictionary views

USER_INDEXES and **USER_IND_COLUMNS**.

Remember that no database object's name, such as an index, cannot be longer than 30 characters.

To drop the index, simply issue the **DROP INDEX** command.

```
DROP INDEX stu_phone_i
```

Index dropped.

- b)** Create a composite index on the **FIRST_NAME** and **LAST_NAME** columns of the **STUDENT** table. Drop the index when you have finished.

ANSWER: There are two possible solutions for creating a composite index using the `FIRST_NAME` and `LAST_NAME` columns.

A composite, or concatenated, index is an index that consists of more than one column. Depending on how you access the table, you need to order the columns in the index accordingly.

To determine the best column order in the index, determine the selectivity of each column. This means determining how many distinct values each column has. You also need to determine what types of queries to write against the table. All this information helps you choose the best column order for the index.

587

588

SOLUTION 1

The index is created in the order `FIRST_NAME`, `LAST_NAME`.

```
CREATE INDEX stu_first_last_name_i  
ON student(first_name, last_name)
```

This index is used in a SQL statement if you refer in the `WHERE` clause to either both columns or the `FIRST_NAME` column. Oracle can access the index if

the WHERE clause lists the leading column of the index. The leading column, also called the leading edge, of the aforementioned index is the FIRST_NAME column. If the WHERE clause of a SQL statement lists only the LAST_NAME column, the SQL statement cannot access the index. For example, the next two WHERE clauses do not use the index.

```
WHERE last_name = 'Smith'  
WHERE last_name LIKE 'Sm%'
```

SOLUTION 2

The index is created in the order LAST_NAME, FIRST_NAME. The LAST_NAME column is the leading column of the index.

```
CREATE INDEX stu_last_first_name_i  
ON student(last_name, first_name)
```

This index is used in a SQL statement if you query both columns or the LAST_NAME column. If a WHERE clause in a SQL statement lists only the FIRST_NAME column, Oracle does not use the index because it is not the leading column of the index.

COMPOSITE INDEXES VERSUS INDIVIDUAL INDEXES

An alternative to using a composite index is to create two separate indexes: one for the FIRST_NAME column and one for the LAST_NAME column.

```
CREATE INDEX stu_first_name_i  
ON student(first_name)  
Index created.
```

```
CREATE INDEX stu_last_name_i  
ON student(last_name)  
Index created.
```

A SQL statement with one of the columns in the WHERE clause uses the appropriate index. In a case where both columns are used in the WHERE clause, Oracle typically merges the two indexes together to retrieve the rows. Why, then, have concatenated indexes at all? A composite index outperforms individual column indexes, provided that all the columns are referenced in the WHERE clause.

SKIP SCAN

A feature called skip scan allows the skipping of the leading edge of an index. During a skip scan, the B-tree

index is probed for the distinct values of the leading edge column. Ideally in such a scenario, the column has only very few distinct values. In the case of the `FIRST_NAME` and `LAST_NAME` columns, there are many different values, which means the Oracle optimizer will probably not use the skip scan feature.

A skip scan is not as fast as an index lookup because for each distinct leading edge value, the index needs to be probed. But if no leading edge index exists, the skip scan feature can allow queries to use the composite index instead of reading the entire table.

588

A second benefit of the skip scan feature is the reduced need for indexes; fewer indexes require less storage space and therefore result in better performance of DML statements.

589



Skip scan is not supported for bitmap and function-based indexes.

The database designer, together with the application developer, decides how to structure the indexes to make them most useful, based on the SQL statements issued. Make sure to verify that Oracle actually uses the index

in your statement; you can do this with the help of an explain plan, as discussed in [Chapter 18](#).

Assume that on a given table, you create a composite index on columns A, B, and C, in this order. To make use of the index, specify in the WHERE clause either column A; columns A and B; columns A, B, and C; or columns A and C. Queries listing column C only, or B only, or B and C only do not use the index because they are not leading edge columns.

To determine the best order, again think about the types of queries issued and the selectivity of each column. The following three indexes cover all the query possibilities. This solution requires the least amount of storage and offers the best overall performance.

```
CREATE INDEX test_table_a_b_c ON
test_table(a, b, c)
CREATE INDEX test_table_b_c ON
test_table(b, c)
CREATE INDEX test_table_c ON
test_table(c)
```

Your queries may be able to take advantage of the skip scan feature, and you may not need to build as many indexes. You must test your statements carefully to ensure adequate performance.

- c) Create an index on the DESCRIPTION column of the COURSE table. Note that queries against the table often use the UPPER function. Drop the index after you successfully create it.

ANSWER: A function-based index is created on the DESCRIPTION column.

```
CREATE INDEX crse_description_i
  ON course(UPPER(description))
```

A function-based index stores the indexed values and uses the index on the following SELECT statement, which retrieves the course number for the course called Hands-On Windows. If you don't know in what case the description was entered into the COURSE table, you might want to apply the UPPER function to the column.

```
SELECT course_no, description
  FROM course
 WHERE UPPER(description) = 'HANDS-
ON WINDOWS'
```

Any query that modifies a column with a function in the WHERE clause does not make use of an index unless you create a function-based index.

An index like the following cannot be used for the previously issued SQL statement.

```
CREATE INDEX crse_description_i
ON course(description)
```

To restore the schema to its previous state, drop the index.

```
DROP INDEX crse_description_i
Index dropped.
```

589

590

- d) Execute the following SQL statements. Explain the reason for the error.

```
CREATE TABLE test (col1 NUMBER)
CREATE INDEX test_col1_i ON
test(col1)
DROP TABLE test
DROP INDEX test_col1_i
```

ANSWER: Dropping a table automatically drops any associated index. There is no need to drop the index separately.

```
DROP INDEX test_col1_i
*
ERROR at line 1:
ORA-01418: specified index does not exist
```

- e) Would a B-tree index work on a frequently accessed column with few distinct values? Explain.

ANSWER: It may be advantageous to create a B-tree index even on a low-selectivity column.

Assume that you have an EMPLOYEE table with a column named GENDER that you consider indexing. Also assume that 90 percent of your employees are male and 10 percent are female. You frequently query for female employees. In this case, the index is helpful and improves the performance of your query. A query for male employees will probably perform a full table scan because this is more efficient than looking up all the values in the index; the Oracle optimizer (discussed in [Chapter 18](#)) makes the decision regarding the best access path.

- f) List the advantages and disadvantages of indexes on performance.

ANSWER: Advantages: Adding an index on a table increases the performance of SQL statements using the indexed column(s) in the WHERE clause. This assumes that only a small percentage

of the rows are accessed. If you access many rows in the table, accessing the entire table via a full table scan probably yields better performance. Indexes on the foreign key columns also improve locking.

Disadvantages: Adding indexes may increase the time required for insert, update, and delete operations because the index needs to be updated. Indexes also require additional disk space.

- g) Assume that an index exists on the column ENROLL_DATE in the ENROLLMENT table. Change the following query so it uses that index.

```
SELECT student_id, section_id,  
       TO_CHAR(enroll_date, 'DD-MON-  
YYYY')  
FROM enrollment  
WHERE TO_CHAR(enroll_date, 'DD-  
MON-YYYY') = '12-MAR-2007'
```

ANSWER: When you modify an indexed column with a function, such as the function TO_CHAR in the WHERE clause, the SQL statement is not able to access the index. The exception is when you create a function-based index on the column. In this case, you do not need a function-based index.

The SQL statement is changed so it does not modify the indexed column with a function.
[Chapter 5](#), “Date and Conversion Functions,” discusses the dangers of using TO_CHAR with a DATE column in the WHERE clause.

```
SELECT student_id, section_id,  
       TO_CHAR(enroll_date, 'DD-MON-  
YYYY')  
FROM enrollment  
WHERE enroll_date = TO_DATE('12-  
MAR-2007', 'DD-MON-YYYY')
```

590

591

Lab 13.1 Quiz

In order to test your progress, you should be able to answer the following questions.

- 1) For the following query, choose which index or indexes, if any, probably yield the best performance.

```
SELECT student_id, last_name,  
       employer, phone  
FROM student  
WHERE employer = 'FGIC'  
AND phone = '201-555-5555'
```

_____ a) Index on employer

_____ **b)** Index on phone

_____ **c)** Index in the order employer, phone

_____ **d)** Index in the order phone, employer

_____ **e)** No index

2) You should always index as many columns as possible.

_____ **a)** True

_____ **b)** False

3) Frequently queried columns and foreign keys should almost always be indexed.

_____ **a)** True

_____ **b)** False

4) The ROWID is the fastest way to access a row.

_____ **a)** True

_____ **b)** False

5) The following query uses the single-column B-tree index on the ZIP column of the INSTRUCTOR table.

```
SELECT instructor_id, last_name,  
first_name, zip  
FROM instructor  
WHERE zip IS NULL
```

_____ a) True

_____ b) False

6) The following SQL statement benefits from an index on the column INSTRUCTOR_ID.

```
UPDATE instructor  
SET phone = '212-555-1212'  
WHERE instructor_id = 123
```

_____ a) True

_____ b) False

ANSWERS APPEAR IN APPENDIX A.

591

592

LAB 13.2 Sequences

LAB OBJECTIVES

After this lab, you will be able to:

- ▶ Create Sequences
- ▶ Use Sequences

Sequences are Oracle database objects that allow you to generate unique integers. Recall the STUDENT table with the primary key column STUDENT_ID. The value of STUDENT_ID is a *surrogate key* or an *artificial key* generated from a sequence. This key is useful to the system but usually has no meaning for the user, is not subject to changes, and is never NULL.

Assume that a student is uniquely identified by first name, last name, and address. These columns are called the *alternate key*. If you choose these columns as the primary key, imagine a scenario in which a student's name or address changes. A large number of updates are required in many tables because all the foreign key columns need to be changed; these changes involve a lot of customized programming. Instead, you can create a surrogate key column and populated it with a sequence. This surrogate key is not subject to change, and the users rarely see this column.

Sequences ensure that no user gets the same value from the sequence, thus guaranteeing unique values for primary keys. Sequences are typically incremented by 1, but other increments can be specified. You can also start sequences at a specific number.

Because you still need to enforce your users' business rules and prevent duplicate student entries, consider creating a unique constraint on the alternate key.

Creating a Sequence

The syntax for creating sequences is as follows.

```
CREATE SEQUENCE sequencename  
  [INCREMENT BY integer]  
  [START WITH integer]  
  [CACHE integer|NOCACHE]  
  [MAXVALUE integer|NOMAXVALUE]  
  [MINVALUE integer|NOMINVALUE]  
  [CYCLE|NOCYCLE]  
  [ORDER|NOORDER]
```

592

593

To create a sequence named `STUDENT_ID_SEQ_NEW`, issue the `CREATE SEQUENCE` command.

```
CREATE SEQUENCE student_id_seq_new  
START WITH 1 NOCACHE
```

Sequence created.

Basing the name of the sequence on the name of the column for which you want to use it is helpful for identification, but it does not associate the sequence with a particular column or table. The `START WITH` clause starts the sequence with the number 1.

The NOCACHE keyword indicates that sequence numbers should not be kept in memory, so that when the system shuts down, you do not lose any cached numbers. However, losing numbers is not a reason for concern because there are many more available from the sequence. It is useful to leave the sequence numbers in the cache only if you access the sequence frequently. If you don't specify a CACHE choice, by default the first 20 numbers are cached.

The MAXVALUE and MINVALUE parameters determine the minimum and maximum range values of the sequence; the defaults are NOMAXVALUE and NOMINVALUE. The ORDER option, which is the default, ensures that the sequence numbers are generated in order of request. The CYCLE parameter recycles the numbers after it reaches the maximum or minimum value, depending on whether it's an ascending or descending sequence; it restarts at the minimum and maximum values, respectively. The default value is NOCYCLE.

Using Sequence Numbers

To increment the sequence and display the unique number, use the NEXTVAL pseudocolumn. The

following SQL statement takes the next value from the sequence. Because the sequence was just created and starts with the number 1, it takes the number 1 as the first available value.

```
SELECT student_id_seq_new.NEXTVAL
FROM dual
NEXTVAL
-----
1

1 row selected.
```

Typically, you use NEXTVAL in INSERT and UPDATE statements. To display the current value of the sequence after it is incremented, use the CURRVAL pseudocolumn.

Altering a Sequence

The ALTER SEQUENCE command allows you to change the properties of a sequence, such as the increment value, min and max values, and cache option. The syntax of the ALTER SEQUENCE command is as follows.

```
ALTER SEQUENCE sequencename
  [INCREMENT BY integer]
  [MAXVALUE integer|NOMAXVALUE]
  [MINVALUE integer|NOMINVALUE]
```

593

594

[CACHE integer | NOCACHE]
[CYCLE | NOCYCLE]
[ORDER | NOORDER]

To restart a sequence at a lower number, you can drop and re-create the sequence. Any GRANTs to other users of the sequence must be reissued. For more on the GRANT command, see [Chapter 15](#), “Security.”

Alternatively, you can issue an ALTER SEQUENCE command to reduce MAXVALUE to a number close to the current sequence value. Also make sure the sequence is set to restart from the beginning by setting the CYCLE value and, if desired, the appropriate MINVALUE. Get NEXTVAL from the sequence to restart the sequence. Remember to reset the MAXVALUE value to a larger number when you are done.

Renaming a Sequence

As with other Oracle objects, you can rename a sequence with the RENAME command. When you rename an object, dependent objects become invalid. Therefore, review the Dependencies tab in SQL Developer or query the data dictionary view USER_DEPENDENCIES to determine the impact of your change.

```
RENAME student_id_seq_new TO  
student_id_seq_newname
```

Using Sequence Values

The NEXTVAL and CURRVAL pseudocolumns can be used in the following SQL constructs.

- ▶ VALUES clause of an INSERT statement
- ▶ SET clause of an UPDATE statement
- ▶ SELECT list (unless it is part of a subquery, view, or materialized view)
- ▶ SELECT list of a subquery in an INSERT statement

Sequence values are not allowed in the following statements.

- ▶ Subquery of a SELECT, UPDATE, or DELETE statement
- ▶ SELECT statement containing DISTINCT, GROUP BY, ORDER BY, UNION, UNION ALL, INTERSECT, or MINUS
- ▶ WHERE clause of a SELECT statement

- 595

FIGURE 13.7 Details tab for the STUDENT_ID_SEQ index in SQL Developer

[illegible]

The Details tab shows the creation date of the sequence, along with the last number used on the sequence; most of this information is from the ALL_SEQUENCES data dictionary view.

The Dependencies tab shows whether any object refers to the sequence. An example of a reference to a sequence is a trigger that selects from this sequence.

LAB 13.2 EXERCISES

- a) Describe the effects of the following SQL statement on the sequence SECTION_ID_SEQ.

```
INSERT INTO section
  (section_id, course_no,
   section_no,
   start_date_time, location,
   instructor_id, capacity,
   created_by,
   created_date, modified_by,
   modified_date)
VALUES
  (section_id_seq.NEXTVAL, 122, 6,
   TO_DATE('15-MAY-2007', 'DD-MON-
   YYYY'), 'R305',
   106, 10, 'ARISCHERT',
   SYSDATE, 'ARISCHERT', SYSDATE)
```

b) Write a SQL statement that increments the sequence STUDENT_ID_SEQ_NEW with NEXTVAL and then issue a ROLLBACK command. Determine the effect on the sequence number.

c) Drop the sequence STUDENT_ID_SEQ_NEW.

595

596

LAB 13.2 EXERCISE ANSWERS

a) Describe the effects of the following SQL statement on the sequence SECTION_ID_SEQ.

```
INSERT INTO section
  (section_id, course_no,
   section_no,
   start_date_time, location,
   instructor_id, capacity,
   created_by,
   created_date, modified_by,
   modified_date)
VALUES
  (section_id_seq.NEXTVAL, 122, 6,
   TO_DATE('15-MAY-2007', 'DD-MON-
   YYYY'), 'R305',
   106, 10, 'ARISCHERT',
   SYSDATE, 'ARISCHERT', SYSDATE)
```


ANSWER: The sequence is accessible from within an INSERT statement. The next value is incremented, and this value is inserted into the table.

AUTOMATING PRIMARY KEY CREATION WITH TRIGGERS

You can automatically increment a sequence and insert the primary key value whenever you insert a new row in a table. This can be accomplished when you write a trigger. Following is the code for a trigger associated with the SECTION table. The trigger fires upon INSERT to the SECTION table. It checks whether a value for the SECTION_ID column is supplied as part of the INSERT statement. If not, it retrieves the value from the SECTION_ID_SEQ sequence and holds the value in the correlation variable :new.SECTION_ID. This value is then inserted into the SECTION_ID column.

```
CREATE OR REPLACE TRIGGER
section_trg_bir
BEFORE INSERT ON section
FOR EACH ROW
BEGIN
```

```
IF :new.SECTION_ID IS NULL THEN
  SELECT section_id_seq.NEXTVAL
  INTO :new.SECTION_ID
  FROM DUAL;
END IF;
END;
/
```

The next command shows the primary key column `SECTION_ID` not listed as part of the `INSERT` statement. The command is successful; it does not return an error message indicating that the primary key column `SECTION_ID` is missing.

```
INSERT INTO section
(course_no, section_no,
instructor_id, created_by,
created_date,
modified_by, modified_date)
VALUES
(20, 99, 109, 'Alice', SYSDATE,
'Alice', SYSDATE)
1 row created.
```

A subsequent `SELECT` statement queries the newly inserted row and displays the automatically created `SECTION_ID` value.

596

597

```
SELECT section_id, course_no, section_no, created_date
FROM section
WHERE course_no = 20
AND section_no = 99
SECTION_ID  COURSE_NO  SECTION_NO  CREATED_D
-----
161         20         99  12-SEP-08

1 row selected.
```

SQL Developer can create triggers to generate primary key values from sequences. When you right-click on a table's node and then select Trigger, you will see the Create (PK from Sequence) menu option (see [Figure 13.8](#)). Oracle lets you choose the sequence to generate the primary key value. The DDL tab shows the dynamically created trigger PL/SQL source code.

FIGURE 13.8 SQL Developer menu option for creating a primary key sequence trigger



For more information on triggers and PL/SQL, refer to *Oracle PL/SQL by Example*, 4th Edition, by Benjamin Rosenzweig and Elena Silvestrova Rakhimov (Prentice Hall, 2008) or the *Oracle Application Developer's Guide—Fundamentals Manual*.

- b) Write a SQL statement that increments the sequence STUDENT_ID_SEQ_NEW with NEXTVAL and then issue a ROLLBACK command. Determine the effect on the sequence number.

ANSWER: After a sequence is incremented, the ROLLBACK command does not restore the number.

If you haven't already done so, create the sequence with the CREATE SEQUENCE student_id_seq_new command. Then retrieve the next number from the sequence.

```
SELECT student_id_seq_new.NEXTVAL
FROM dual
NEXTVAL
-----
2

1 row selected.

ROLLBACK
Rollback complete.

SELECT student_id_seq_new.NEXTVAL
FROM dual
NEXTVAL
-----
3

1 row selected.
```

597

598

If there are any gaps in the primary key sequence numbers, it really doesn't matter because the numbers have no meaning to the user, and there are many more numbers available from the sequence. One of the unique properties of sequences is that no two users receive the same number.

You can see information about the sequence in the USER_SEQUENCES data dictionary view. Here the LAST_NUMBER column indicates the last used number of the sequence. Alternatively, you can see this information in SQL Developer's Details tab for the sequence.

```
SELECT sequence_name, last_number, cache_size
FROM user_sequences
WHERE sequence_name = 'STUDENT_ID_SEQ_NEW'
```

SEQUENCE_NAME	LAST_NUMBER	CACHE_SIZE
STUDENT_ID_SEQ_NEW	3	0

```
1 row selected.
```

You can obtain the current number of the sequence by using CURRVAL, provided that the sequence was incremented by the user's session.

```
SELECT student_id_seq_new.CURRVAL
FROM dual
CURRVAL
-----
3
1 row selected.
```

c) Drop the sequence STUDENT_ID_SEQ_NEW.

ANSWER: As with other database objects, you use the DROP command to drop a sequence.

```
DROP SEQUENCE student_id_seq_new
```

Sequence dropped.

598

599

Lab 13.2 Quiz

In order to test your progress, you should be able to answer the following questions.

1) Sequences are useful for generating unique values.

_____ a) True

_____ b) False

2) A student's Social Security number is a good choice for a primary key value instead of a sequence.

_____ a) True

_____ b) False

3) The default increment of a sequence is 1.

_____ a) True

_____ b) False

4) When you drop a table, the associated sequence is also dropped.

_____ **a)** True

_____ **b)** False

5) The following statement creates a sequence named EMPLOYEE_ID_SEQ, which starts at the number 1000.

```
CREATE SEQUENCE employee_id_seq  
START WITH 1000
```

_____ **a)** True

_____ **b)** False

ANSWERS APPEAR IN APPENDIX A.

599

600

LAB 13.3 Views

LAB OBJECTIVES

After this lab, you will be able to:

- ▶ Create, Alter, and Drop Views
- ▶ Understand the Data Manipulation Rules for Views

A view is a virtual table that consists of columns and rows, but it is only the SELECT statement that is stored, not a physical table with data. A view's SELECT query may reference one or multiple tables, called *base tables*. The base tables are typically actual tables or other views.

Advantages of Views

Views simplify the writing of queries. You can query a single view instead of writing a complicated SQL statement that joins many tables. The complexity of the underlying SQL statement is hidden from the user and contained only in the view.

Views are useful for security reasons because they can hide data. The data retrieved from a view can show only certain columns if you list those columns in the SELECT list of the query. You can also restrict the view to display specific rows with the WHERE clause of the query.

In a view, you can give a column a different name from the one in the base table. Views may be used to isolate an application from a change in the definition of the base tables. Rather than change the program, you can make changes to the view.

A view looks just like any other table. You can describe and query the view and also issue INSERT, UPDATE,

and DELETE statements to a certain extent, as you will see when performing the exercises in this lab.

Creating a View

The simplified syntax for creating a view is as follows.

```
CREATE [OR REPLACE] [FORCE|NOFORCE]
VIEW viewname
[(column_alias[, column_alias]...)]
AS query
[WITH CHECK OPTION|WITH READ ONLY
[CONSTRAINT constraintname]]
```

600

601

The following statements create a view called **COURSE_NO_COST** and describe the new view.

```
CREATE OR REPLACE VIEW course_no_cost AS
SELECT course_no, description, prerequisite
FROM course
View created.
```

```
SQL> DESC course_no_cost
```

Name	Null?	Type
COURSE_NO	NOT NULL	NUMBER(8)
DESCRIPTION	NOT NULL	VARCHAR2(50)
PREREQUISITE		NUMBER(8)

The **COURSE_NO_COST** view hides a number of columns that exist in the **COURSE** table. You do not see the **COST** column or the **CREATED_DATE**,

CREATED_BY, MODIFIED_DATE, and MODIFIED_BY columns. The main purpose of this view is security. You can grant access just to the view COURSE_NO_COST instead of to the COURSE table itself. For more information on granting access privileges to database objects, see [Chapter 15](#).

Using Column Aliases

The following statement demonstrates a view with column names different from the column names in the base tables. Here the view named STUD_ENROLL shows a list of the STUDENT_ID, the last name of the student in capital letters, and the number of classes the student is enrolled in. The column STUDENT_ID from the STUDENT table is renamed in the view to STUD_ID, using a column alias. When a column contains an expression such as a function, a column alias is required. The two expressions in the STUD_ENROLL view, namely the student last name in caps and the count of classes enrolled, are therefore aliased.

```
CREATE OR REPLACE VIEW stud_enroll
AS
SELECT s.student_id stud_id,
       UPPER(s.last_name) last_name,
       COUNT(*) num_enrolled
```

```
FROM student s, enrollment e
WHERE s.student_id = e.student_id
GROUP BY s.student_id,
UPPER(s.last_name)
```

The OR REPLACE keyword is useful if the view already exists. It allows you to replace the view with a different SELECT statement without having to drop the view first. This also means you do not have to re-grant privileges to the view; the rights to the view are retained by those who have already been granted access privileges.

The following example shows an alternate SQL statement for naming columns in a view, whereby the view's columns are listed in parentheses after the view name.

```
CREATE OR REPLACE VIEW stud_enroll
    (stud_id, last_name,
num_enrolled) AS
SELECT s.student_id,
    UPPER(s.last_name),
    COUNT(*)
FROM student s, enrollment e
WHERE s.student_id = e.student_id
GROUP BY s.student_id,
UPPER(s.last_name)
```

601

602

Altering a View

You use the ALTER VIEW command to recompile a view if it becomes invalid. This can occur after you alter one of the base tables. The syntax of the ALTER VIEW statement is as follows.

```
ALTER VIEW viewname COMPILE
```

The ALTER VIEW command allows for additional syntax options not mentioned. These options let you create primary or unique constraints on views. However, these constraints are not enforced and do not maintain data integrity, and an index is never built because they can only be created in DISABLE NOVALIDATE mode. These constraint types are primarily useful with *materialized views*, a popular data warehousing feature that allows you to physically store pre-aggregated results and/or joins for speedy access. Unlike the views discussed in this chapter, materialized views result in physical data stored in tables.

Renaming a View

The RENAME command allows you to change the name of a view.

```
RENAME stud_enroll TO stud_enroll2
```

All underlying constraints and granted privileges remain intact. However, any objects that use this view (perhaps another view or a PL/SQL procedure, package, or function) become invalid and need to be compiled.

Dropping a View

To drop a view, you use the DROP VIEW command. The following statement drops the STUD_ENROLL2 view.

```
DROP VIEW stud_enroll2
```

View dropped.

602

603

Lab 13.3 Exercise Answers

- a) Create a view called LONG_DISTANCE_STUDENT with all the columns in the STUDENT table plus the CITY and STATE columns from the ZIPCODE table. Exclude students from New York, New Jersey, and Connecticut.
- b) Create a view named CHEAP_COURSE that shows all columns of the COURSE table where the course cost is 1095 or less.

- c)** Issue the following INSERT statement. What do you observe when you query the CHEAP_COURSE view?

```
INSERT INTO cheap_course
(course_no, description, cost,
created_by, created_date,
modified_by,
modified_date)
VALUES
(900, 'Expensive', 2000,
'ME', SYSDATE, 'ME', SYSDATE)
```

- d)** Drop the views named LONG_DISTANCE_STUDENT and CHEAP_COURSE.
- e)** Using the following statement, create a table called TEST_TAB and build a view over it. Then, add a column to the table and describe the view. What do you observe? Drop the table and view after you complete the exercise.

```
CREATE TABLE test_tab
(coll1 NUMBER)
```

- f)** Create a view called BUSY_STUDENT, based on the following query. Update the number of

enrollments for STUDENT_ID 124 to five through the BUSY_STUDENT view. Record your observation.

```
SELECT student_id, COUNT(*)  
  FROM enrollment  
 GROUP BY student_id  
HAVING COUNT(*) > 2
```

- g) Create a view that lists the addresses of students. Include the columns STUDENT_ID, FIRST_NAME, LAST_NAME, STREET_ADDRESS, CITY, STATE, and ZIP. Using the view, update the last name of STUDENT_ID 237 from Frost to O'Brien. Then update the state for the student from NJ to CT. What do you notice for the statements you issue?

LAB 13.3 EXERCISE ANSWERS

- a) Create a view called LONG_DISTANCE_STUDENT with all the columns in the STUDENT table plus the CITY and STATE columns from the ZIPCODE table. Exclude students from New York, New Jersey, and Connecticut.

ANSWER: To select all columns from the STUDENT table, use the wildcard symbol. For the columns CITY and STATE in the view, join to the ZIPCODE table. With this view definition, you see only records where the state is not equal to New York, Connecticut, or New Jersey.

```
CREATE OR REPLACE VIEW
long_distance_student AS
SELECT s.*, z.city, z.state
FROM student s, zipcode z
WHERE s.zip = z.zip
```

603

```
AND state NOT IN
('NJ', 'NY', 'CT')
```

604

View created.

You can issue a query against the view or describe the view. You can restrict the columns and/or the rows of the view.

```
SELECT state, first_name, last_name
FROM long_distance_student
ST FIRST_NAME          LAST_NAME
-----
MA James E.            Norman
MA George              Kocka
...
OH Phil                Gilloon
MI Roger              Snow

10 rows selected.
```


You might want to validate the view by querying for students living in New Jersey.

```
SELECT *  
  FROM long_distance_student  
 WHERE state = 'NJ'  
no rows selected
```

The query finds no students living in New Jersey because the view's defining query excludes those records.

- b)** Create a view named CHEAP_COURSE that shows all columns of the COURSE table where the course cost is 1095 or less.

ANSWER: Creating a view as follows restricts the rows to courses with a cost of 1095 or less.
CREATE OR REPLACE VIEW cheap_course
AS

```
SELECT *  
  FROM course  
 WHERE cost <= 1095
```

- c)** Issue the following INSERT statement. What do you observe when you query the CHEAP_COURSE view?

```
INSERT INTO cheap_course
(course_no, description, cost,
created_by, created_date,
modified_by,
modified_date)
VALUES
(900, 'Expensive', 2000,
'ME', SYSDATE, 'ME', SYSDATE)
```

ANSWER: You can insert records through the view, violating the view's defining query condition.

A cost of 2000 is successfully inserted into the COURSE table through the view, even though this is higher than 1095, which is the defining condition of the view.

You can query CHEAP_VIEW to see if the record is there. The course was successfully inserted in the underlying COURSE base table, but it does not satisfy the view's definition and is not displayed.

```
SELECT course_no, cost
FROM cheap_course
COURSE_NO    COST
-----
135          1095
230          1095
240          1095

3 rows selected.
```

604

605

A view's WHERE clause works for any query, but not for DML statements. The course number 900 is not visible through the CHEAP_COURSE view, but insert, update, or delete operations are permitted despite the conflicting WHERE condition. To change this security-defying behavior, create the view with the WITH CHECK OPTION constraint. But first undo the INSERT statement with the ROLLBACK command because any subsequent DDL command, such as the creation of a view, automatically commits the record.

ROLLBACK

Rollback complete.

```
CREATE OR REPLACE VIEW
cheap_course AS
SELECT *
  FROM course
 WHERE cost <= 1095
WITH CHECK OPTION CONSTRAINT
check_cost
View created.
```

It is a good habit to name constraints. You understand the benefit of well-named constraints

when you query the Oracle data dictionary or when you violate constraints with data manipulation statements.

The following error message appears when insertions, updates, and deletions issued against a view violate the view's defining query. The previous INSERT statement would now be rejected, with the error ORA-01402 view WITH CHECK OPTION where-clause violation.

What happens if you attempt to insert a record with a value of NULL for the course cost? Again, Oracle rejects the row because the condition is not satisfied. The NULL value is not less than or equal to 1095.

VIEW CONSTRAINTS

You can enforce constraints in a variety of ways: The underlying base tables automatically ensure data integrity, or you can use the WITH CHECK OPTION. You can also avoid any data manipulation on the view by using the READ ONLY syntax option. The following statement creates a read-only view named COURSE_V.

```
CREATE OR REPLACE VIEW course_v AS
```

```
SELECT course_no, description,  
       created_by, created_date,  
       modified_by, modified_date
```

605

```
FROM course  
WITH READ ONLY CONSTRAINT  
course_v_read_check
```

606

View created.

- d) Drop the views named
LONG_DISTANCE_STUDENT and
CHEAP_COURSE.

ANSWER: Just like other operations on data objects, the DROP keyword removes a database object from the database.

```
DROP VIEW long_distance_student
```

View dropped.

```
DROP VIEW cheap_course
```

View dropped.



Remember that any DDL operation, such as the creation of a view, cannot be rolled back, and any prior DML operations, such as inserts, updates, and deletes, are automatically committed.

- e) Using the following statement, create a table called TEST_TAB and build a view over it. Then add a column to the table and describe the view. What do you observe? Drop the table and view after you complete the exercise.

```
CREATE TABLE test_tab  
  (col1 NUMBER)
```

ANSWER: The view does not show the newly added column.

```
CREATE OR REPLACE VIEW  
test_tab_view AS  
SELECT *  
  FROM test_tab
```

View created.

After the table creation, the view is created. Here, the name TEST_TAB_VIEW is used. Then you add an additional column to the TEST_TAB table; here it is named COL2.

```
ALTER TABLE test_tab  
  ADD (col2 NUMBER)
```

Table altered.

A subsequently issued DESCRIBE of the view reveals an interesting fact.

```
SQL> DESC test_tab_view
```

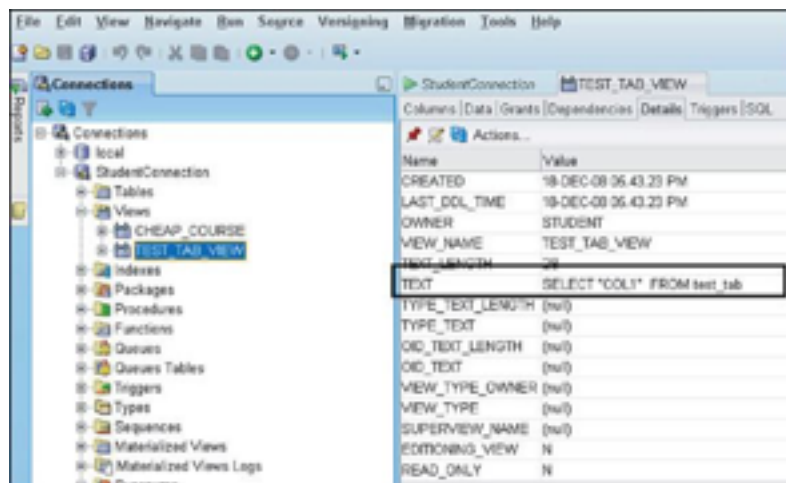
Name	Null?	Type
-----	-----	-----
COL1		NUMBER

Where is the new column that was added?

Whenever a view is created with the wildcard (*) character, Oracle stores the individual column names in the definition of the view. When you review the Details tab of the view (see [Figure 13.9](#)) in SQL Developer, you see that the view consists only of one column. This is the view's definition at the time of view creation. Note the column is also listed with enclosed quotation marks, just in case of mixed case column names.

606
607

FIGURE 13.9 The TEXT box shows the view's query definition



Alternatively, you can query Oracle's data dictionary view USER_VIEWS.

```
SELECT text
  FROM user_views
 WHERE view_name = 'TEST_TAB_VIEW'
TEXT
-----
SELECT "COL1"
  FROM test_tab

1 row selected.
```

You need to reissue the creation of the view statement for the view to include the new column.

```
CREATE OR REPLACE VIEW
test_tab_view AS
SELECT *
  FROM test_tab
```

View created.

Now, when a DESCRIBE is issued on the view, the new column is included.

```
SQL> DESC test_tab_view
```

Name	Null?	Type
COL1		NUMBER
COL2		NUMBER

607

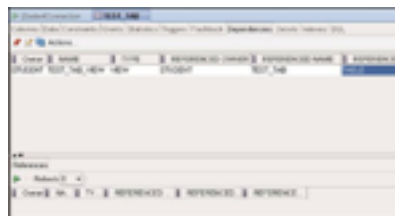
COMPILING A VIEW

You can use the ALTER VIEW command to define, modify, or drop view constraints. Also, the command ALTER VIEW viewname COMPILE command explicitly compiles the view to make sure it is valid. A view may become invalid if the underlying table is altered or dropped. If you use Oracle 11g, the view remains valid if you modify a column in the base table that's not being used by this view.

DEPENDENCIES TAB

You can see in the Dependencies tab for a given table whether any objects access the table; this helps you assess the impact of changes. [Figure 13.10](#) shows the dependencies of the TEST_TAB table. As you see, TEST_TAB_VIEW depends on this table, and if you click the Dependencies tab for the TEST_TAB_VIEW, you find that the TEST_TAB table is referenced in the view.

FIGURE 13.10 Dependencies tab



ACCESSING AN INVALID VIEW

When you access an invalid view, Oracle attempts to recompile it automatically. It is useful to explicitly compile to the view, as shown in the next command, to ensure that there are no problems with the view after you make database changes.

```
ALTER VIEW test_tab_view COMPILE
View altered.
```

Drop the no-longer-needed table and notice the effect on the view.

```
DROP TABLE test_tab
Table dropped.
ALTER VIEW test_tab_view COMPILE
Warning: View altered with
compilation errors.
```

Following is an attempt to retrieve data from the invalid view; Oracle returns an error message to the user, indicating that the view exists. However, it is currently invalid because the underlying objects were altered or dropped. Any subsequent attempt to access the view or to compile it returns an error.

```
SELECT *
FROM test_tab_view
```

608

609

```
ERROR at line 2:  
ORA-04063: view  
"STUDENT.TEST_TAB_VIEW" has errors
```

Drop the view to restore the STUDENT schema to its previous state.

```
DROP VIEW test_tab_view  
View dropped.
```

FORCING THE CREATION OF A VIEW

If a view's base tables do not exist or if the creator of the view doesn't have privileges to access the view, the creation of the view fails. The following example shows the creation of the view named TEST, based on a nonexistent SALES table.

```
CREATE VIEW test AS  
SELECT *  
FROM sales  
ERROR at line 3:  
ORA-00942: table or view does not  
exist
```

If you want to create the view, despite its being invalid, you can create it with the FORCE option; the default in the CREATE VIEW syntax is NOFORCE. This FORCE option is useful if you need to create the view and you

add the referenced table later or if you expect to obtain the necessary privileges to the referenced object shortly.

```
CREATE OR REPLACE FORCE VIEW test
AS
SELECT *
FROM sales
```

Warning: View created with compilation errors.

The view, though invalid, now exists in the database.

- f) Create a view called BUSY_STUDENT, based on the following query. Update the number of enrollments for STUDENT_ID 124 to five through the BUSY_STUDENT view. Record your observation.

```
SELECT student_id, COUNT(*)
FROM enrollment
GROUP BY student_id
HAVING COUNT(*) > 2
```

ANSWER: The UPDATE operation fails. Data manipulation operations on a view impose a number of restrictions.

To create the view, you need to give the COUNT(*) expression a column alias; otherwise, the following error occurs.

ERROR at line 2:

ORA-00998: must name this expression with a column alias

609

610

```
CREATE OR REPLACE VIEW
busy_student AS
SELECT student_id, COUNT(*)
enroll_num
FROM enrollment
GROUP BY student_id
HAVING COUNT(*) > 2
```

View created.

You can now attempt to update the ENROLLMENT table using the view with the following UPDATE statement.

```
UPDATE busy_student
SET enroll_num = 5
WHERE student_id = 124
```

ORA-01732: data manipulation operation not legal on this view

DATA MANIPULATION RULES ON VIEWS

For a view to be updatable, it needs to conform to a number of rules. The view cannot contain any of the following.

- ▶ An expression (for example, `TO_DATE(enroll_date)`)
- ▶ An aggregate function
- ▶ A set operator, such as `UNION`, `UNION ALL`, `INTERSECT`, or `MINUS`
- ▶ The `DISTINCT` keyword
- ▶ The `GROUP BY` clause
- ▶ The `ORDER BY` clause

Special rules apply to views that contain join conditions, as shown in exercise g.

- g) Create a view that lists the addresses of students. Include the columns `STUDENT_ID`, `FIRST_NAME`, `LAST_NAME`, `STREET_ADDRESS`, `CITY`, `STATE`, and `ZIP`.

Using the view, update the last name of STUDENT_ID 237 from Frost to O'Brien. Then update the state for the student from NJ to CT. What do you notice for the statements you issue?

ANSWER: Not all updates to views containing joins are allowed. The update of the last name is successful, but the update of the STATE column is not.

```
CREATE OR REPLACE VIEW
student_address AS
SELECT student_id, first_name,
last_name,
       street_address, city, state,
s.zip szip,
z.zip zzip
FROM student s, zipcode z
WHERE s.zip = z.zip
View created.
```

Now update the last name to O'Brien by using the following statement. To indicate a single quotation mark, prefix it with another quotation mark.

```
UPDATE student_address
       SET last_name = 'O''Brien'
       WHERE student_id = 237
```

1 row updated.

610

Because the test was successful, roll back the update to retain the current data in the table.

611

```
ROLLBACK
```

Rollback complete.

You can update the data in the underlying base table STUDENT. Now update the column STATE in the base table ZIPCODE through the STUDENT_ADDRESS view.

```
UPDATE student_address
```

```
SET state = 'CT'
```

```
WHERE student_id = 237
```

ORA-01779: cannot modify a column which maps to a nonkey-preserved table

JOIN VIEWS AND DATA MANIPULATION

Understanding the concept of key-preserved tables is essential to understanding the restrictions on join views. A table is considered key preserved if every key of the table can also be a key of the result of the join. In this case, the STUDENT table is the key-preserved, or child, table.

For a join view to be updatable, the DML operation may affect only the key-preserved table (also known as the child base table), and the child's primary key must be included in the view's definition. In this case, the child table is the STUDENT table, and the primary key is the STUDENT_ID column.

If you are in doubt regarding which table is the key-preserved table, query the Oracle data dictionary table USER_UPDATABLE_COLUMNS. The result shows you which columns are updatable. Also, the STUDENT table's ZIP column is updatable, but the ZIP column from the ZIPCODE table is not. Only the STUDENT table's ZIP column (aliased as SZIP) is considered key preserved.

```
SELECT column_name, updatable
  FROM user_updatable_columns
 WHERE table_name = 'STUDENT_ADDRESS'

```

COLUMN_NAME	UPD
STUDENT_ID	YES
FIRST_NAME	YES
LAST_NAME	YES
STREET_ADDRESS	YES
CITY	NO
STATE	NO
SZIP	YES
ZZIP	NO

8 rows selected.

The data dictionary is covered in greater detail in [Chapter 14](#), “The Data Dictionary, Scripting, and Reporting.” If you need to manipulate key-preserved data through a view, you overcome this limitation by using an INSTEAD OF trigger. This trigger works only against views and allows you to manipulate data based on the code within the trigger. The INSTEAD OF trigger fires in place of your issued INSERT, UPDATE, or DELETE command against the view. For example, if you execute an INSERT command against the view, the statement may actually perform an UPDATE instead. The view’s associated INSTEAD OF trigger code can perform any type of data manipulation against one or multiple tables. You create these powerful INSTEAD OF triggers by using Oracle PL/SQL, as covered in great detail in *Oracle PL/SQL by Example*, 4th Edition, by Benjamin Rosenzweig and Elena Silvestrova Rakhimov (Prentice Hall, 2008).

611

612

Lab 13.3 Quiz

In order to test your progress, you should be able to answer the following questions.

- 1) Views are useful for security, for simplifying the writing of queries, and for hiding data complexity.

_____ a) True

_____ b) False

2) Under what circumstances can views become invalid? Select all that apply.

_____ a) The data type of a column referenced in the view changes.

_____ b) One of the underlying base tables is dropped.

_____ c) Views never become invalid; they automatically recompile.

3) Identify the error in the following view definition.

```
CREATE OR REPLACE VIEW my_student
  (studid, slname, szip) AS
SELECT student_id, last_name, zip
  FROM student
 WHERE student_id BETWEEN 100 AND
200
```

_____ a) Line 1

_____ b) Line 2

_____ c) Line 4

_____ d) Lines 1, 2, and 4

_____e) No error

- 4) An UPDATE to the STATE column in the ZIPCODE table is permitted using the following view.

```
CREATE OR REPLACE VIEW my_zipcode
AS
SELECT zip, city, state,
created_by,
       created_date, modified_by,
       TO_CHAR(modified_date, 'DD-MON-
YYYY') modified_date
FROM zipcode
```

_____a) True

_____b) False

- 5) Views provide security by restricting access to specific rows and/or columns of a table.

_____a) True

_____b) False

- 6) A column in a view may have a different name than in the base table.

_____a) True

_____ **b) False**

ANSWERS APPEAR IN APPENDIX A.

612