SQL

Structured Query Language (SQL) is the database language by the use of which we can perform certain operations on the existing database and also we can use this language to create a database. SQL uses certain commands like Create, Drop, Insert etc. to carry out the required tasks.

SQL is a language to:

Create databases and the objects within them.
Store data in those databases.
Change and analyze that data.
Get that data back out in reports, web pages, or virtually any other use
Imaginable.

These SQL commands are mainly categorized into four categories as:

1. DDL – Data Definition Language
2. DQL – Data Query Language
3. DML – Data Manipulation Language
4. DCL – Data Control Language
5. TCL- Transaction Control Language

1. **DDL (Data Definition Language)**: DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.
   Examples of DDL commands:
   - CREATE – is used to create the database or its objects (like table, index, function, views, store procedure and triggers).
   - DROP – is used to delete objects from the database.
   - ALTER-is used to alter the structure of the database.
   - TRUNCATE–is used to remove all records from a table, including all spaces allocated for the records are removed.
   - COMMENT –is used to add comments to the data dictionary.
   - RENAME –is used to rename an object existing in the database.
2. **DQL (Data Query Language)** :
   DQL statements are used for performing queries on the data within schema objects. The purpose of DQL Command is to get some schema relation based on the query passed to it.

   Example of DQL:
   - SELECT – is used to retrieve data from the database.
3. **DML (Data Manipulation Language)**: The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements.
   Examples of DML:
   - INSERT – is used to insert data into a table.
   - UPDATE – is used to update existing data within a table.
   - DELETE – is used to delete records from a database table.
   - MERGE -- is the combination of three INSERT, DELETE and UPDATE statements. So if there is a Source table and a Target table that are to be merged, then with the help of MERGE statement, all the three operations (INSERT, UPDATE, DELETE) can be performed at once.
4. **DCL (Data Control Language)**: DCL includes commands such as GRANT and REVOKE which mainly deals with the rights, permissions and other controls of the database system.
   Examples of DCL commands:
   - GRANT-gives user's access privileges to database.
   - REVOKE-withdraw user's access privileges given by using the GRANT command.
5. **TCL (transaction Control Language)**: TCL commands deals with the transaction within the database.
   Examples of TCL commands:
   - COMMIT– commits a Transaction.
   - ROLLBACK– rollbacks a transaction in case of any error occurs.
   - SAVEPOINT–sets a savepoint within a transaction.
   - SET TRANSACTION–specify characteristics for the transaction.

Rules Are Meant to be followed
SQL is a fairly strict language in terms of syntax rules, but it remains simple and flexible enough to support a variety of programming styles. This section discusses some of the basic rules governing SQL statements.

Uppercase or Lowercase
It is a matter of personal taste about the case in which SQL statements are submitted to the database. When interacting with literal values, case does matter.

Metadata about different database objects is stored by default in uppercase in the data dictionary. If you query a database dictionary table to return a list of tables owned by the HR schema, it is likely that the table names returned are stored in uppercase. This does not mean that a table cannot be created with a lowercase name; it can be. It is just more common and the default behavior of the Oracle server to create and store tables, columns, and other database object metadata in uppercase in the database dictionary.

Statement Terminators
Semicolons are generally used as SQL statement terminators. SQL*Plus always requires a statement terminator, and usually a semicolon is used. A single SQL statement or even groups of associated statements are often saved as script files for future use. Individual statements in SQL scripts are commonly terminated by a line break (or carriage return) and a forward slash on the next line, instead of a semicolon. You can create a SELECT statement, terminate it with a line break, include a forward slash to execute the

statement, and save it in a script file. The script file can then be called from within SQL*Plus. Note that SQL Developer does not require a statement terminator if only a single statement is present, but it will not object if one is used. It is good practice to always terminate your SQL statements with a semicolon. Several examples of SQL*Plus statements follow:

```
select country_name, country_id, location_id from countries;
select city, location_id,
state_province, country_id
from locations
/
```

The first example of code demonstrates two important rules. First, the statement is terminated by a semicolon. Second, the entire statement is written on one line. It is entirely acceptable for a SQL statement to either be written on one line or to span multiple lines as long as no words in the statement span multiple lines. The second sample of code demonstrates a statement that spans three lines that is terminated by a new line and executed with a forward slash.

SQL statements may be submitted to the database in either lowercase or uppercase. You must pay careful attention to case when interacting with character literal data and aliases.
Asking for a column called JOB_ID or job_id returns the same column, but asking for rows where the JOB_ID value is PRESIDENT is different from asking for rows where the JOB_ID value is President. Character literal data should always be treated in a case-sensitive manner.

Indentation, Readability, and Good Practice

Consider the following query:

```
select city, location_id,
state_province, country_id
from locations
/
```

This example highlights the benefits of indenting your SQL statement to enhance the readability of your code. The Oracle server does not object if the entire statement is written on one line without indentation. It is good practice to separate different clauses of the SELECT statement onto different lines. When an expression in a clause is particularly complex, it is often better to separate that term of the statement onto a new line. When developing SQL to meet your reporting needs, the process
is often iterative. The SQL interpreter is far more useful during development if complex expressions are isolated on separate lines, since errors are usually thrown in the format of: “ERROR at line X:” This makes the debugging process much simpler.

Chapter 1: Data Retrieval Using the SQL SELECT Statement

SELECT statement is to extract or retrieve data stored in relational tables.
SELECT statement never alters information stored in the database. Instead, it provides a read-only method of extracting information.

Capability of the SELECT statement: projection, selection, and joining.
Projection refers to the restriction of attributes (columns) selected from a relation or table. You would request just required columns from the table. This restriction of columns is called *projection*.

Selection refers to the restriction of the tuples or rows selected from a relation (table). It is often not desirable to retrieve every row from a table. Tables may contain many rows and, instead of asking for all of them, selection provides a means to restrict the rows returned.
Joining, as a relational concept, refers to the interaction of tables with each other in a query to retrieve data from both tables.

Syntax of the Primitive SELECT Statement

In its most primitive form, the SELECT statement supports the projection of columns and the creation of arithmetic, character, and date expressions. It also facilitates the elimination of duplicate values from the results set.

SELECT *|{[DISTINCT] *column|expression* [*alias*],…}
FROM *table*;

A SELECT statement always comprises two or more clauses. The two mandatory clauses are the SELECT clause and the FROM clause.

SELECT *
FROM *table;*

```
SELECT * FROM Employee;
```

The second form of the basic SELECT statement has the same FROM clause as the first form, but the SELECT clause is different:

SELECT {[DISTINCT] *column|expression* [*alias*],…}
FROM *table;*

This SELECT clause can be simplified into two formats:

SELECT *column1 (possibly other columns or expressions)* [alias optional]
OR
SELECT DISTINCT *column1 (possibly other columns or expressions)* [alias
optional]

An *alias* is an alternative name for referencing a column or expression. Aliases are typically used for displaying output in a user-friendly manner.

```
SELECT REGION_NAME
FROM REGIONS;
```

Using the DISTINCT keyword allows duplicate rows to be eliminated from the results set.
An important feature of the DISTINCT keyword is the elimination of duplicate values from *combinations* of columns.

```
select distinct department_id
from employees;
```

SQL Expressions and Operators

The general form of the SELECT statement introduced the notion that columns and expressions are selectable. An expression is usually made up of an operation being performed on one or more column values. The operators that can act upon column values to form an expression depend on the data type of the column.

They are the four arithmetic operators (addition, subtraction, multiplication, and division) for numeric columns.
The concatenation operator for character or string columns.
The addition and subtraction operators for date and timestamp columns.

Arithmetic Operators

Precedence of Arithmetic Operators

Highest ( ) Brackets or parentheses
Medium / Division
Medium ∗ Multiplication
Lowest − Subtraction
Lowest + Addition

Expression and Column Aliasing

An *alias* is an alternate name for a column or an expression. Aliases are especially useful with expressions or calculations and may be implemented in several ways. Quotation marks are necessary for two reasons. First, this alias is made up of more than one word. Second, case preservation of an alias is only possible if the alias is double quoted.
The AS keyword is optional since it is also possible to use a space before specifying an alias, as discussed earlier. Use of the AS keyword does, however, improve the readability of SQL statements, and the authors believe it is a good SQL coding habit to form.

Character and String Concatenation Operator

The double pipe symbols || represent the character *concatenation* operator. This operator is used to join character expressions or columns together to create a larger character expression. Columns of a table may be linked to each other or to strings of literal characters to create one resultant character expression. Concatenation operator is flexible enough to be used multiple times and almost anywhere in a character expression.

Literals and the DUAL Table

Literal values in expressions are a common occurrence. These values refer to numeric, character, or date and time values found in SELECT clauses that do not originate from any database object. Concatenating character literals to existing column data can be useful.

To ensure relational consistency, Oracle offers a clever solution to the problem of using the database to evaluate expressions that have nothing to do with any tables or columns. To get the database to evaluate an expression, a syntactically legal SELECT statement must be submitted. Oracle solves the problem of relational interaction with the database operating on literal expressions by offering a special table called DUAL.

It contains one column called DUMMY of character data type.

You can execute the query: `SELECT * FROM DUAL`, and the data value "X" is returned as the contents of the DUMMY column.

The DUAL table allows literal expressions to be selected from it for processing and returns the expression results in its single row. It is exceptionally useful since it enables a variety of different processing requests to be made from the database.

Testing complex expressions during development, by querying the dual table, is an effective method to evaluate whether these expressions are working correctly. Literal expressions can be queried from any table, but remember that the expression will be processed for every row in the table.

```
select 'literal'||'processing using the REGIONS table'
from regions;
```

The preceding statement will return four lines in the results set, since there are four rows of data in the REGIONS table.

Two Single Quotes or the Alternative Quote Operator

The literal character strings concatenated so far have been singular words prepended and appended to column expressions. These character literals are specified using single quotation marks. For example:

```
select 'I am a character literal string'
```

What about character literals that contain single quotation marks? Plurals pose a particular problem for character literal processing. Consider the following statement:

```
select 'Plural's have one quote too many' from dual;
```

executing this statement causes an Oracle error to be generated. So, how are words that contain single quotation marks dealt with?

There are essentially two mechanisms available. The most popular of these is to add an additional single quotation mark next to each naturally occurring single quotation mark in the character string.

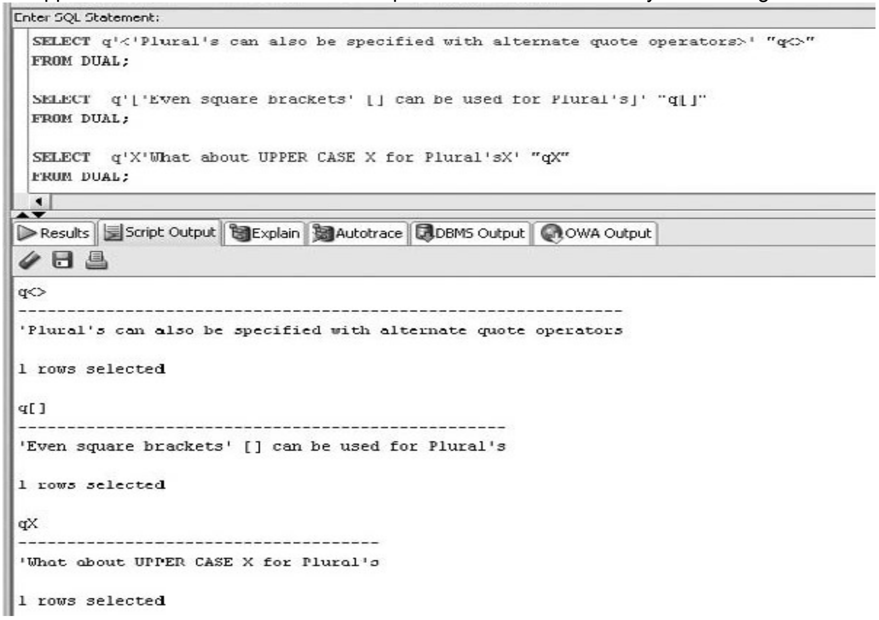
Previous error is avoided by replacing the character literal `'Plural's` with the literal `'Plural''s`.

Oracle offers a way to deal with this type of character literal in the form of the alternative **quote (q) operator**. Notice that the problem is that Oracle chose the single quote characters as the special pair of symbols that enclose or wrap any other character literal. These character-enclosing symbols could have been anything other than single quotation marks.

Bearing this in mind, consider the alternative quote (q) operator. The q operator enables you to choose from a set of possible pairs of wrapping symbols for character literals as alternatives to the single quote symbols. The options are any single-byte or multibyte character or the four brackets: (round brackets), {curly braces}, [square brackets], or <angle brackets>. Using the q operator, the character delimiter can effectively be changed from a single quotation mark to any other character.

The syntax of the alternative quote operator is as follows:

q'*delimiter*'character literal which may include the single quotes *delimiter*' where *delimiter* can be any character or bracket. The first and second examples show the use of angle and square brackets as character delimiters, while the third example demonstrates how an uppercase "X" has been used as the special character delimiter symbol through the alternative quote operator.



```
SELECT q'<'Plural's can also be specified with alternate quote operators>' "q<>"
FROM DUAL;

SELECT  q'['Even square brackets' [] can be used for Plural's]' "q[]"
FROM DUAL;

SELECT  q'X'What about UPPER CASE X for Plural'sX' "qX"
FROM DUAL;
```

```
q<>
---------------------------------------------------------------
'Plural's can also be specified with alternate quote operators

1 rows selected

q[]
--------------------------------------------------
'Even square brackets' [] can be used for Plural's

1 rows selected

qX
-------------------------------------
'What about UPPER CASE X for Plural's

1 rows selected
```

NULL Is Nothing

Both the number zero and a blank space are different from null since they occupy space. *Null* refers to an absence of data. A row that contains a null value lacks data for that column. Null is formally defined as a value that is unavailable, unassigned, unknown, or inapplicable. In other words, the rules of engaging with null values need careful examination. Failure to heed the special treatment that null values require will almost certainly lead to an error, or worse, an inaccurate answer.

Null values may be a tricky concept to come to grips with. The problem stems from the absence of null on a number line. It is not a real, tangible value that can be related to the physical world.

Null is a placeholder in a nonmandatory column until some real data is stored in its place. Until then, beware of conducting arithmetic with null columns.

Not Null and Nullable Columns

Tables store rows of data that are divided into one or more columns. These columns have names and data types associated with them. Some of them are constrained by database rules to be mandatory columns. It is compulsory for some data to be stored in the

NOT NULL columns in each row. When columns of a table, however, are not compelled by the database constraints to hold data for a row, these columns run the risk of being empty.

.

Nullable is a term sometimes used to describe a column that is allowed to store null values.

Any arithmetic calculation with a NULL value always returns NULL.

The character concatenation operators ignore null, whilst the arithmetic operations involving null values always result in null.

Experimenting with Expressions and the DUAL Table

In this step-by-step exercise a connection is made using SQL Developer as the HR user. Use expressions and operators to answer three questions related to the SELECT statement:

Question 1: It was demonstrated earlier how the number of days for which staff were employed in a job could be calculated. For how many years were staff employed while fulfilling these job roles and what were their EMPLOYEE_ID, JOB_ID, START_DATE, and END_DATE values? Alias the expression column in your query with the alias Years Employed. Assume that a year consists of 365.25 days.

```
select employee_id, job_id, start_date, end_date,
((end_date-start_date) + 1)/365.25 "Years Employed"
from job_history;
```

Question 2: Query the JOBS table and return a single expression of the form The Job Id for the <job_title's> job is: <job_id>. Take note that the job_title should have an apostrophe and an "s" appended to it to read more naturally. A sample of this output for the organization president is: "The Job Id for the President's job is: AD_PRES." Alias this column expression as "Job Description" using the AS keyword.

```
select 'The Job Id for the '||job_title||'''s job is: '||job_id
AS "Job Description"
from jobs;
```

Question 3: Using the DUAL table, calculate the area of a circle with radius 6000 units, with pi being approximately 22/7. Use the formula: Area = pi × radius ×radius. Alias the result as "Area."

```
select (22/7) * (6000 * 6000) Area
from dual
```

✓ TWO-MINUTE DRILL

List the Capabilities of SQL SELECT Statements

❑ The three fundamental operations that SELECT statements are capable of are projection, selection, and joining.

❑ Projection refers to the restriction of columns selected from a table. Using projection, you retrieve only the columns of interest and not every possible column.

❑ Selection refers to the extraction of rows from a table. Selection includes the further restriction of the extracted rows based on various criteria or conditions. This allows you to retrieve only the rows that are of interest and not every row in the table.

❑ Joining involves linking two or more tables based on common attributes. Joining allows data to be stored in third normal form in discrete tables, instead of in one large table.

❑ An unlimited combination of projections, selections, and joins provides the language to extract the relational data required.

❑ A structural definition of a table can be obtained using the DESCRIBE command.

❑ Columns in tables store different types of data using various data types, the most common of which are NUMBER, VARCHAR2, DATE, and TIMESTAMP.

❑ The data type NUMBER(*x*,*y*) implies that numeric information stored in this column can have at most *x* digits, but at least y of these digits must appear on the right hand side of the decimal point.

❑ The DESCRIBE command lists the names, data types, and nullable status of all columns in a table.

❑ Mandatory columns are also referred to as NOT NULL columns

❑ The syntax of the primitive SELECT clause is as follows:
SELECT *|{[DISTINCT] column|expression [alias],…}

❑ The SELECT statement is also referred to as a SELECT query and comprises at least two clauses, namely the SELECT clause and the FROM clause.

❑ The SELECT clause determines the *projection* of columns. In other words, the SELECT clause specifies which columns are included in the results returned.

❑ The asterisk (*) operator is used as a wildcard symbol to indicate all columns. So, the statement SELECT * FROM ACCOUNTS returns all the columns available in the ACCOUNTS table.

❑ The FROM clause specifies the source table or tables from which items are selected.

❑ The DISTINCT keyword preceding items in the SELECT clause causes duplicate combinations of these items to be excluded from the returned results set.

❑ SQL statements should be terminated with a semicolon. As an alternative, a new line can be added after a statement and a forward slash can be used to execute the statement.

❑ SQL statements can be written and executed in lowercase or uppercase. Be careful when interacting with character literals since these are case-sensitive.

❑ Arithmetic operators and the string concatenation operator acting on column and literal data form the basis of SQL expressions.

❑ Expressions and regular columns may be aliased using the AS keyword or by leaving a space between the column or expression and the alias.

❑ If an alias contains multiple words or the case of the alias is important, it must be enclosed in double quotation marks.

❑ Naturally occurring single quotes in a character literal can be selected by making use of either an additional single quote per naturally occurring quote or the alternative quote operator.

❑ The DUAL table is a single column and single row table that is often used to evaluate expressions that do not refer to specific columns or tables.

❑ Columns which are not governed by a NOT NULL constraint have the potential to store null values and are sometimes referred to as nullable columns.

❑ NULL values are not the same as a blank space or zero. NULL values refer to an absence of data. Null is defined as a value that is unavailable, unassigned, unknown, or inapplicable.

❑ Caution must be exercised when working with null values since arithmetic with a null value always yields a null result.

SELF TEST
Read all the choices carefully because there might be more than one correct answer. Choose all the correct answers for each question.

1. Which query creates a projection of the DEPARTMENT_NAME and LOCATION_ID columns from the DEPARTMENTS table? (Choose the best answer.)
A. SELECT DISTINCT DEPARTMENT_NAME, LOCATION_ID FROM DEPARTMENTS;
B. SELECT DEPARTMENT_NAME, LOCATION_ID FROM DEPARTMENTS;
C. SELECT DEPT_NAME, LOC_ID FROM DEPT;
D. SELECT DEPARTMENT_NAME AS “LOCATION_ID” FROM DEPARTMENTS;

2. After describing the EMPLOYEES table, you discover that the SALARY column has a data type of NUMBER(8,2). Which SALARY value(s) will not be permitted in this column? (Choose all that apply.)
A. SALARY=12345678
B. SALARY=123456.78
C. SALARY=12345.678
D. SALARY=123456
E. SALARY=12.34

3. After describing the JOB_HISTORY table, you discover that the START_DATE and END_DATE columns have a data type of DATE. Consider the expression END_DATE-START_DATE. (Choose two correct statements.)
A. A value of DATE data type is returned.
B. A value of type NUMBER is returned.
C. A value of type VARCHAR2 is returned.
D. The expression is invalid since arithmetic cannot be performed on columns with DATE data types.
E. The expression represents the days between the END_DATE and START_DATE less one day.

4. The DEPARTMENTS table contains a DEPARTMENT_NAME column with data type VARCHAR2(30). (Choose two true statements about this column.)
A. This column can store character data up to a maximum of 30 characters.
B. This column must store character data that is at least 30 characters long.
C. The VARCHAR2 data type is replaced by the CHAR data type.
D. This column can store data in a column with data type VARCHAR2(50) provided that the contents are at most 30 characters long.

5. Which statement reports on unique JOB_ID values from the EMPLOYEES table? (Choose all that apply.)
A. SELECT JOB_ID FROM EMPLOYEES;
B. SELECT UNIQUE JOB_ID FROM EMPLOYEES;
C. SELECT DISTINCT JOB_ID, EMPLOYEE_ID FROM EMPLOYEES;
D. SELECT DISTINCT JOB_ID FROM EMPLOYEES;

6. Choose the two illegal statements. The two correct statements produce identical results. The two illegal statements will cause an error to be raised:
A. SELECT DEPARTMENT_ID|| ' represents the '||DEPARTMENT_NAME||' Department' as "Department Info"
FROM DEPARTMENTS;
B. SELECT DEPARTMENT_ID|| ' represents the ||DEPARTMENT_NAME||' Department' as "Department Info"
FROM DEPARTMENTS;
C. select department_id|| ' represents the '||department_name||' Department' "Department Info" from departments;
D. SELECT DEPARTMENT_ID represents the DEPARTMENT_NAME Department as "Department Info"
FROM DEPARTMENTS;

7. Which expressions do not return NULL values? (Choose all that apply.)
A. select ((10 + 20) * 50) + null from dual;
B. select 'this is a '||null||'test with nulls' from dual;
C. select null/0 from dual;
D. select null||'test'||null as “Test” from dual;

8. Choose the correct syntax to return all columns and rows of data from the EMPLOYEES table.
A. select all from employees;
B. select employee_id, first_name, last_name, first_name, department_id from employees;
C. select % from employees;
D. select * from employees;
E. select *.* from employees;

9. The following character literal expression is selected from the DUAL table:
SELECT 'Coda''s favorite fetch toy is his orange ring' FROM DUAL; (Choose the result that is returned.)
A. An error would be returned due to the presence of two adjacent quotes
B. Coda's favorite fetch toy is his orange ring
C. Coda''s favorite fetch toy is his orange ring
D. 'Coda''s favorite fetch toy is his orange ring'

10. There are four rows of data in the REGIONS table. Consider the following SQL statement:

SELECT '6 * 6' “Area” FROM REGIONS;
How many rows of results are returned and what value is returned by the Area column?(Choose the best answer.)
A. 1 row returned, Area column contains value 36
B. 4 rows returned, Area column contains value 36 for all 4 rows
C. 1 row returned, Area column contains value 6 * 6
D. 4 rows returned, Area column contains value 6 * 6 for all 4 rows
E. A syntax error is returned.

LAB QUESTION
1. Obtain structural information for the PRODUCT_INFORMATION and ORDERS tables.
2. Select the unique SALES_REP_ID values from the ORDERS table. How many different sales representatives have been assigned to orders in the ORDERS table?
3. Create a results set based on the ORDERS table that includes the ORDER_ID, ORDER_DATE, and ORDER_TOTAL columns. Notice how the ORDER_DATE output is formatted differently from the START_DATE and END_DATE columns in the HR.JOB_ID table.
4. The PRODUCT_INFORMATION table stores data regarding the products available for sale in a fictitious IT hardware store. Produce a set of results that will be useful for a sales person. Extract product information in the format <PRODUCT_NAME> with code: <PRODUCT_ID> has status of: <PRODUCT_STATUS>. Alias the expression as “Product.” The results should provide the LIST_PRICE, the MIN_PRICE, the difference between LIST_PRICE, and MIN_PRICE aliased as “Max Actual Savings,” along with an additional expression that takes the difference between LIST_PRICE and MIN_PRICE and divides it by the LIST_PRICE and then multiplies the total by 100. This last expression should be aliased as “Max Discount %.”
5. Calculate the surface area of the Earth using the DUAL table. Alias this expression as “Earth's Area.” The formula for calculating the area of a sphere is: 4□r2. Assume, for this example, that the earth is a simple sphere with a radius of 3,958.759 miles and that □ is 22/7.

SELF TEST ANSWERS

1. ✹✓ B. A projection is an intentional restriction of the columns returned from a table.

✹⮟ **A** is eliminated since the question has nothing to do with duplicates, distinctiveness, or uniqueness of data. **C** incorrectly selects nonexistent columns called DEPT_NAME and LOC_ID from a nonexistent table called DEPT. **D** returns just one of the requested columns: DEPARTMENT_NAME. Instead of additionally projecting the LOCATION_ID column from the DEPARTMENTS table, it attempts to alias the DEPARTMENT_NAME column as LOCATION_ID.

2. ✹✓ A and **C.** Columns with NUMBER(8,2) data type can store, at most, eight digits; of which, at most, two of those digits are to the right of the decimal point. Although **A** and **C** are the correct answers, note that since the question is phrased in the negative, these values are NOT allowed to be stored in such a column. **A** is not allowed because it contains eight whole number digits, but the data type is constrained to store six whole number digits and two fractional digits. **C** is not allowed since it has three fractional digits and the data type allows a maximum of two fractional digits.
✹⮟ **B, D,** and **E** can legitimately be stored in this data type and, therefore, are the incorrect answers to this question. **D** shows that numbers with no fractional part are legitimate values for this column, as long as the number of digits in the whole number portion does not exceed six digits.

3. ✹✓ B and **E.** The result of arithmetic between two date values represents a certain number of days.

✹⮟ **A, C,** and **D** are incorrect. It is a common mistake to expect the result of arithmetic between two date values to be a date as well, so **A** may seem plausible, but it is false.

4. ✹✓ A and **D.** The scale of the VARCHAR2 data type, specified in brackets, determines its maximum capacity for storing character data as mentioned by **A**. If a data value that is at most 30 characters long is stored in any data type, it can also be stored in this column as stated by **D**.
✹⮟ **B** is incorrect because it is possible to store character data of any length up to 30 characters in this column. **C** is false, since the CHAR data type exists in parallel with the VARCHAR2 data type.

5. ✹✓ D. Unique JOB_ID values are projected from the EMPLOYEES table by applying the DISTINCT keyword to just the JOB_ID column.
✹⮟ **A, B,** and **C** are eliminated since **A** returns an unrestricted list of JOB_ID values including duplicates; **B** makes use of the UNIQUE keyword in the incorrect context; and **C** selects the distinct combination of JOB_ID and EMPLOYEE_ID values. This has the effect of returning all the rows from the EMPLOYEES table since the EMPLOYEE_ID column contains unique values for each employee record. Additionally, **C** returns two columns, which is not what was originally requested.

6. ✹✓ B and **D** represent the two illegal statements that will return syntax errors if they are executed. This is a tricky question because it asks for the illegal statements and not the legal statements. **B** is illegal because it is missing a single quote enclosing the character literal “represents the.” **D** is illegal because it does not make use of single quotes to enclose its character literals.
✹⮟ **A** and **C** are the legal statements and, therefore, in the context of the question, are the incorrect answers. **A** and **C** appear to be different since the case of the SQL statements are different and **A** uses the alias keyword AS, whereas **C** just leaves a space between the expression and the alias. Yet both **A** and **C** produce identical results.

7. ✹✓ B and **D** do not return null values since character expressions are not affected in the same way by null values as arithmetic expressions. **B** and **D** ignore the presence of null values in their expressions and return the remaining character literals.
✹⮟ **A** and **C** return null values because any arithmetic expression that involves a null will return a null.

8. ✹✓ D. An asterisk is the SQL operator that implies that all columns must be selected from a table.

✹⮟ **A, B, C,** and **E** are incorrect. **A** uses the ALL reserved word but is missing any column specification and will, therefore, generate an error. **B** selects some columns but not all columns and, therefore, does not answer the question. **C** and **E** make use of illegal selection operators.

9. ✹✓ B. The key to identifying the correct result lies in understanding the role of the single quotation marks. The entire literal is enclosed by a pair of quotes to avoid the generation of an error. The two adjacent quotes are necessary to delimit the single quote that appears in literal **B**.
✹⮟ **A, C,** and **D** are incorrect. **A** is eliminated since no error is returned. **C** inaccurately returns two adjacent quotes in the literal expression and **D** returns a literal with all the quotes still present. The Oracle server removes the quotes used as character delimiters after processing the literal.

10. ✹✓ D. The literal expression '6 * 6' is selected once for each row of data in the REGIONS table.

✹⮟ **A, B, C,** and **E** are incorrect. **A** returns one row instead of four and calculates the product 6 * 6. The enclosing quote operators render 6 * 6 a character literal and not a numeric literal that can be calculated. **B** correctly returns four rows but incorrectly evaluates the character literal as a numeric literal. **C** incorrectly returns one row instead of four and **E** is incorrect, because the given SQL statement can be executed.

LAB ANSWER

1. The DESCRIBE command gives us the structural description of a table.

2.

```
select distinct sales_rep_id
from orders;
```

3.

```
select order_id, order_date, order_total
from orders;
```

4.

```
select product_name||' with code: '||product_id'||' has status of:'||order_
status AS Product
select list_price – min_price AS "Max Actual Savings"
Select ((list_price-min_price)/list_price) * 100 AS "Max Discount %"
```

5.

```
select (4 * (22/7) * (3958.759 * 3958.759)) AS "Earth's Area"
from dual;
```

Chapter 2: Restricting and Sorting Data

Limiting the columns retrieved by a SELECT statement is known as *projection* and restricting the rows returned is known as *selection.* The *WHERE* clause specifies one or more conditions that the Oracle server evaluates to restrict the rows returned by the statement. ORDER BY clause provides data sorting capabilities.

Limit the Rows Retrieved by a Query
Selection is actualized using the WHERE clause of the SELECT statement. Conditions that restrict the dataset returned take many forms and operate on columns as well as expressions. Only those rows in a table that conform to these conditions are returned. Conditions restrict rows using comparison operators in conjunction with columns and literal values. Boolean operators provide a mechanism to specify multiple conditions to restrict the rows returned. Boolean, conditional, concatenation, and arithmetic operators are discussed to establish their order of precedence when they are encountered in a SELECT statement.

The following four areas are investigated:

- The WHERE clause
- Comparison operators
- Boolean operators
- Precedence rules

The WHERE clause
The *WHERE* clause extends the SELECT statement by providing the language to restrict rows returned based on one or more conditions.
The format of the SQL SELECT statement which includes the WHERE clause is:
SELECT *|{[DISTINCT] *column|expression* [*alias*],…}
FROM *table*
[WHERE *condition(s)*];

The square brackets indicate that the WHERE clause is optional. One or more conditions may be simultaneously applied to restrict the result set. A condition is specified by comparing two terms using a conditional operator. These terms may be column values, literals, or expressions.
The *equality* operator is most commonly used to restrict result sets.

```
select country_name
from countries
where region_id=3;

select last_name, first_name from employees
where job_id='SA_REP';
```

Numeric-Based Conditions
Conditions must be formulated appropriately for different column data types. The conditions restricting rows based on numeric columns can be specified in several different ways. The first query specifies the number 10000, while the second encloses the number within single quotes like a character literal. Both formats are acceptable to Oracle since an implicit data type conversion is performed when necessary.

```
select last_name, salary from employees
where salary = 10000;
select last_name, salary from employees
where salary = '10000';
```

A numeric column can be compared to another numeric column in the same row to construct a WHERE clause condition, as the following query demonstrates:

```
select last_name, salary from employees
where salary = department_id;


select last_name, salary from employees
where salary/10 = department_id*10;
```

Character-Based Conditions
Conditions determining which rows are selected based on character data, are specified by enclosing character literals in the conditional clause, within single quotes.

```
select last_name
from employees
where job_id='SA_REP';
```

If you tried specifying the character literal without the quotes, an Oracle error would be raised. Remember that character literal data is case sensitive, so the following WHERE clauses are not equivalent.

```
Clause 1: where job_id=SA_REP
Clause 2: where job_id='Sa_Rep'
Clause 3: where job_id='sa_rep'
```

Character-based conditions are not limited to comparing column values with literals. They may also be specified using other character columns and expressions. The LAST_NAME and FIRST_NAME columns are both specified as VARCHAR2(25) data typed columns. Consider the query:

```
select employee_id, job_id
from employees
where last_name=first_name;
```

Both the LAST_NAME and FIRST_NAME columns appear on either side of the equality operator in the WHERE clause. No literal values are present; therefore no single quote characters are necessary to delimit them.

The following four clauses demonstrate some of the options for character-based conditions:

```
Clause 1: where 'A '||last_name||first_name = 'A King'
Clause 2: where first_name||' '||last_name = last_name||' '||first_name
Clause 3: where 'SA_REP'||'King' = job_id||last_name
Clause 4: where job_id||last_name ='SA_REP'||'King'
```

Date-Based Conditions

DATE columns are useful when storing date and time information. Date literals must be enclosed in single quotation marks just like character data; otherwise an error is raised. When used in conditional WHERE clauses, DATE columns are compared to other DATE columns or to date literals. The literals are automatically converted into DATE values based on the default date format, which is DD-MONRR. If a literal occurs in an expression involving a DATE column, it is automatically converted into a date value using the default format mask. DD represents days, MON represents the first three letters of a month, and RR represents a Year 2000–compliant year (that is, if RR is between 50 and 99, then the Oracle server returns the previous century, else it returns the current century). The full four-digit year, YYYY, can also be specified. Consider the following four SQL statements:

```
Statement 1:
select employee_id from job_history
where start_date = end_date;
Statement 2:
select employee_id from job_history
where start_date = '01-JAN-2001';
Statement 3:
select employee_id from job_history
where start_date = '01-JAN-01';
Statement 4:
select employee_id from job_history
where start_date = '01-JAN-99';
```

The 1st statement tests equality between two DATE columns. Rows that contain the same values in their START_DATE and END_DATE columns will be returned.
Note:- however, that DATE values are only equal to each other if there is an exact match between all their components including day, month, year, hours, minutes, and seconds.

In the 2nd statement, the START_DATE column is compared to the character literal: '01-JAN-2001'. The entire four-digit year component (YYYY) has been specified. This is acceptable to the Oracle server, and all rows in the JOB_HISTORY table with START_DATE column values equal to the first of January 2001 will be returned.

The 3rd statement is equivalent to the second since the literal '01-JAN-01' is converted to the date value 01-JAN-2001. This is due to the RR component being less than 50, so the current (twenty-first) century, 20, is prefixed to the year RR component to provide a century value. All rows in the JOB_HISTORY table with START_DATE column values = 01-JAN-2001 will be returned.

The century component for the literal '01-JAN-99' becomes the previous (twentieth) century, 19, yields a date value of 01-JAN-1999 for the fourth statement, since the RR component, 99, is greater than 50. Rows in the JOB_HISTORY table with START_DATE column values = 01-JAN-1999 will be returned.

Arithmetic using the addition and subtraction operators is supported in expressions involving DATE values.
An expression like: END_DATE – START_ DATE returns a numeric value representing the number of days between START_ DATE and END_DATE.
An expression like: START_DATE + 30 returns a DATE value that is 30 days later than START_DATE.

```
select employee_id from job_history
where start_date + 30 = '31-JAN-99';
```

This query returns rows from the JOB_HISTORY table containing a START_DATE value equal to 30 days before 31-JAN-1999.

Comparison Operators

The *equality* operator is used extensively to illustrate the concept of restricting rows using a WHERE clause. There are several alternative operators that may also be used.
The *inequality* operators like “less than” or “greater than or equal to” may be used to return rows conforming to inequality conditions. The *BETWEEN* operator facilitates range-based comparison to test whether a column value lies between two values. The *IN* operator tests set membership, so a row is returned if the column value tested in the condition is a member of a set of literals. The pattern matching comparison operator *LIKE* is extremely powerful, allowing components of character column data to be matched to literals conforming to a specific pattern. The *IS NULL* operator, which returns rows where the column value contains a null value. These operators may be used in any combination in the WHERE clause.

Equality and Inequality

Testing for *equality* in a condition is both natural and intuitive. Such a condition is formed using the “is equal to” (=) operator. A row is returned if the equality condition is true for that row.
Inequality-based conditions enhance the WHERE clause specification. Range and pattern matching comparisons are possible using inequality and equality operators, but it is often preferable to use the BETWEEN and LIKE operators for these comparisons.
.
For example, the following query may be issued to obtain a list of LAST_NAME and SALARY values for employees who earn more that $5000:

```
select last_name, salary from employees
where salary > 5000;
```

The *composite inequality operators* (made up of more than one symbol) are utilized in the following four clauses:

```
Clause 1: where salary <= 3000;
Clause 2: where salary >= 5000;
Clause 3: where salary <> department_id;
Clause 4: where salary != 4000+department_id;
```

Numeric inequality is naturally intuitive. The comparison of character and date terms, however, is more complex. Testing character inequality is interesting since the strings being compared on either side of the inequality operator are converted to a numeric

representation of its characters. Based on the database character set and NLS (National Language Support) settings, each character string is assigned a numeric value. These numeric values form the basis for the evaluation of the inequality comparison.
Consider the following statement:

```
select last_name from employees
where last_name < 'King';
```

The character literal 'King' is converted to a numeric representation. Assuming a US7ASCII database character set with AMERICAN NLS settings, the literal 'King' is converted into a sum of its ordinal character values: K + i + n + g = (75+105+110+103=393). For each row in the EMPLOYEES table, the LAST_NAME column is similarly converted to a numeric value. If this value is less than 393, then the row is selected. The same process for comparing numeric data using the inequality operators applies to character data. The only difference is that character data is converted implicitly by the Oracle server to a numeric value based on certain database settings.

Inequality comparisons operating on date values follow a similar process to character data. The Oracle server stores dates in an internal numeric format, and these values are compared within the conditions. The second of June of a certain year occurs earlier than the third of June of the same year. Therefore, the numeric value of the date 02-JUN-2008 is less than the numeric value of the date 03-JUN-2008.
Consider the following query:

```
select last_name from employees
where hire_date < '01-JAN-2000';
```

This query retrieves each employee record containing a HIRE_DATE value that is earlier than ‘01-JAN-2000’. Rows with employee HIRE_DATE=31-DEC-1999 will be returned, while rows with employee HIRE_DATE values later than the first of January 2000 will not be returned.

Inequality Operators:-
< Less than
> Greater than
<= Less than or equal to
>= Greater than or equal to
<> Not equal to
!= Not equal to

Range Comparison with the BETWEEN Operator
The BETWEEN operator tests whether a column or expression value falls within a range of two boundary values. The item must be at least the same as the lower boundary value, or at most the same as the higher boundary value, or fall within the range, for the condition to be true.
Suppose you want the last names of employees who earn a salary in the range of $3400 and $4000. A possible solution using the BETWEEN operator is as follows:

```
select last_name from employees
where salary between 3400 and 4000;
```

The AND operator is used to specify multiple WHERE conditions, all of which must be satisfied for a row to be returned. Using the AND operator, the BETWEEN operator is equivalent to two conditions using the “greater than or equal to” and “less than or equal to” operators, respectively.

The preceding SQL statement is equivalent to the following statement

```
select last_name from employees
where salary >= 3400
and salary <= 4000;
```

The BETWEEN operator specify the range condition using the BETWEEN operator. The implication of this equivalence is that the mechanism utilized to evaluate numeric, character, and date operands by the inequality operators is the same for the BETWEEN operator.
The following query tests whether the HIRE_DATE column value is later than 24-JUL-1994 but earlier than 07-JUN-1996:

```
select first_name, hire_date from employees
where hire_date between '24-JUL-1994' and '07-JUN-1996';
```

You are not restricted to specifying literal values as the operands to the BETWEEN operator, since these may be column values and expressions such as the following:

```
select first_name, hire_date from employees
where '24-JUL-1994' between hire_date+30 and '07-JUN-1996';
```

For a row to be returned by this query, the date literal 24-JUL-1994 must fall between the row’s HIRE_DATE column value plus 30 days and the date literal 07-JUN-1996.

Set Comparison with the IN Operator
The *IN* operator tests whether an item is a member of a set of literal values. The set is specified by a comma separating the literals and enclosing them in round brackets.
If the literals are character or date values, then these must be delimited using single quotes. You may include as many literals in the set as you wish.
Consider the following example:

```
select last_name from employees
where salary in (1000,4000,6000);
```

The SALARY value in each row is compared for equality to the literals specified in the set. If the SALARY value equals 1000, 4000, or 6000, the LAST_NAME value for that row is returned.

The Boolean OR operator, is used to specify multiple WHERE conditions, at least one of which must be satisfied for a row to be returned. The *IN* operator is therefore equivalent to a series of OR conditions. The preceding SQL statement may be written using multiple OR condition clauses, for example:

```
select last_name from employees
where salary = 1000
OR salary = 4000
OR salary = 6000;
```

This statement will return an employee’s LAST_NAME if at least one of the WHERE clause conditions is true; it has the same meaning as the previous statement that uses the IN operator.

Testing set membership using the *IN* operator is more succinct than using multiple OR conditions, especially as the number of members in the set increases.

The following two statements demonstrate use of the *IN* operator with DATE and CHARACTER data.

```
select last_name from employees
where last_name in ('King','Garbharran','Ramklass');
select last_name from employees
where hire_date in ('01-JAN-1998','01-DEC-1999');
```

Pattern Comparison with the LIKE Operator

The *LIKE* operator, which is designed exclusively for character data and provides a powerful mechanism for searching for letters or words.

LIKE is accompanied by two wildcard characters: the percentage symbol (%) and the underscore character (_).

The percentage symbol is used to specify zero or more wildcard characters, while the underscore character specifies one wildcard character.

A wildcard may represent any character.

You may be requested to provide a list of employees whose first names begin with the letter “A.”

The following query can be used to provide this set of results:

```
select first_name from employees
where first_name like 'A%';
```

The wildcard characters can appear at the beginning, middle or at the end of the character literal. They can even appear alone as in:

```
where first_name like '%';
```

In this case, every row containing a FIRST_NAME value that is not null will be returned.

Wildcard symbols are not mandatory when using the LIKE operator. In such cases, LIKE behaves as an equality operator testing for exact character matches.

so the following two WHERE clauses are equivalent:

```
where last_name like 'King';
where last_name = 'King';
```

Consider searching for employees whose last names are four letters long, begin with a “K,” have an unknown second letter, and end with an ng.

You may issue the following statement:

```
where last_name like 'K_ng';
```

The two wildcard symbols can be used independently, together, or even multiple times in a single WHERE condition.

What about the scenario when you are searching for a literal that contains a percentage or underscore character? Oracle provides a way to temporarily disable their special meaning and regard them as regular characters using the *ESCAPE* identifier.

The JOBS table contains JOB_ID values that are literally specified with an underscore character, such as SA_MAN, AD_VP, MK_REP, and SA_REP.

Assume there is, in addition, a row in the JOBS table with a JOB_ID of SA%MAN. Notice there is no underscore character in this JOB_ID. How can values be retrieved from the JOBS table if you are looking for JOB_ID values beginning with the characters SA_?

Consider the following SQL statement:

```
select * from jobs
where job_id like 'SA_%';
```

This query will return the rows SA_REP, SA_MAN, and SA%MAN. The requirement in this example is not met since an additional row, SA%MAN, not on forming to the criterion that it begins with the characters SA_, is also returned,

A naturally occurring underscore character may be escaped (or treated as a regular nonspecial symbol) using the ESCAPE identifier in conjunction with an ESCAPE character.

```
select job_id from jobs
where job_id like 'SA\_%' escape '\';
```

retrieves the JOBS table records with JOB_ID values equal to SA_MAN and SA_REP and which conforms to the original requirement.

The ESCAPE identifier instructs the Oracle server to treat any character found after the backslash character as a regular nonspecial symbol with no wildcard meaning.

Traditionally, the ESCAPE character is the backslash symbol, but it does not have to be. The following statement is equivalent to the previous one but uses a dollar symbol as the ESCAPE character instead.

```
select job_id from jobs
where job_id like 'SA$_%' escape '$';
```

The percentage symbol may be similarly escaped when it occurs naturally as character data.

Suppose, there is a requirement to retrieve the row with the hypothetical JOB_ID: SA%MAN introduced earlier. Querying the JOBS table for JOB_ID values such as SA%MAN using the following code results in the records SA_MAN and SA%MAN being returned.

```
select job_id from jobs
where job_id like 'SA\%MAN' ESCAPE '\';
```

Is it possible to use the LIKE comparison operator with numeric data?

Yes. Oracle automatically casts the data into the required data type, if possible. In this case, the numeric SALARY values are momentarily “changed” into character data allowing the use of the LIKE operator to locate matching patterns. The following query locates the required rows:

```
SELECT * FROM EMPLOYEES
WHERE SALARY LIKE '%80%';”
```

Null Comparison with the IS NULL Operator

NULL values inevitably find their way into database tables. It is often required that only those records that contain a NULL value in a specific column are sought. The *IS NULL* operator selects only the rows where a specific column value is NULL.

Testing column values for equality to NULL is performed using the *IS NULL* operator instead of the “is equal to” operator (=).

```
select last_name from employees
where commission_pct is null;
```

This WHERE clause reads naturally and retrieves only the records which contain NULL COMMISSION_PCT values. The query using the “=” operator does not return same output as “IS NULL” operator.

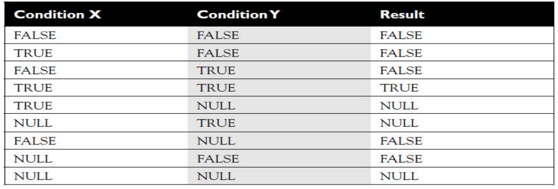
Boolean Operators

Data is restricted using a WHERE clause with a single condition. *Boolean* or *logical* operators enable multiple conditions to be specified in the WHERE clause of the SELECT statement. This facilitates a more refined data extraction capability.

The AND Operator

The *AND* operator merges conditions into one larger condition to which a row must conform to be included in the results set.

AND Operator Truth Table:-



Condition X	Condition Y	Result
FALSE	FALSE	FALSE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	TRUE	TRUE
TRUE	NULL	NULL
NULL	TRUE	NULL
FALSE	NULL	FALSE
NULL	FALSE	FALSE
NULL	NULL	NULL

Boolean operators are defined using *truth tables*. If two conditions specified in a WHERE clause are joined with an AND operator, then a row is tested consecutively for conformance to both conditions before being retrieved. If the row contains a NULL value that causes one of the conditions to evaluate to NULL, then that row is excluded.

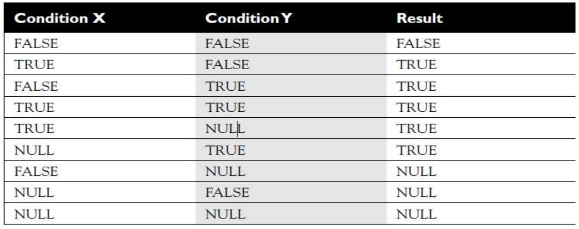
```
select first_name, last_name, commission_pct, hire_date
from employees
where first_name like 'J%'
and commission_pct > 0.1;
```

To specify further mandatory conditions, simply add them and ensure that they are separated by additional AND operators. You can specify as many conditions as you wish.

The OR Operator

The *OR* operator separates multiple conditions, at least one of which must be satisfied by the row selected to warrant inclusion in the results set.

OR operator truth table:-



Condition X	Condition Y	Result
FALSE	FALSE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
TRUE	TRUE	TRUE
TRUE	NULL	TRUE
NULL	TRUE	TRUE
FALSE	NULL	NULL
NULL	FALSE	NULL
NULL	NULL	NULL

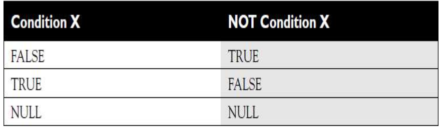
If two conditions specified in a WHERE clause are joined with an OR operator then a row is tested consecutively for conformance to either or both conditions before being retrieved. Conforming to just one of the OR conditions is sufficient for the record to be returned. If it conforms to none of the conditions, the row is excluded since the result is FALSE. A row will only be returned if at least one of the conditions associated with an OR operator evaluates to TRUE.

```
select first_name, last_name, commission_pct, hire_date
from employees
where first_name like 'B%'
or commission_pct > 0.35;
```

Notice that the two conditions are separated by the OR keyword. All employee records with FIRST_NAME values beginning with an uppercase “B” will be returned regardless of their COMMISSION_PCT values, even if they are NULL. All those records having COMMISSION_PCT values greater that 35 percent, regardless of what letter their FIRST_NAME values begin with, are also returned.

The NOT Operator

The *NOT* operator negates conditional operators. A selected row must conform to the logical opposite of the condition in order to be included in the results set. NOT operator negates the comparison operator in a condition, whether it’s an equality, inequality, range based, pattern matching, set membership, or null testing operator.

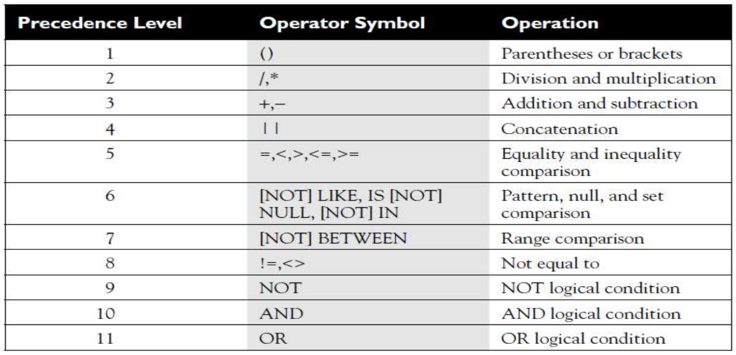


Condition X	NOT Condition X
FALSE	TRUE
TRUE	FALSE
NULL	NULL

```
select first_name, last_name, commission_pct, hire_date
from employees
where first_name not like 'B%'
or not (commission_pct > 0.35);
```

Operator Precedence Hierarchy

Operators at the same level of precedence are evaluated from left to right if they are encountered together in an expression.



Precedence Level	Operator Symbol	Operation
1	()	Parentheses or brackets
2	/,*	Division and multiplication
3	+,–	Addition and subtraction
4	||	Concatenation
5	=,<,>,<=,>=	Equality and inequality comparison
6	[NOT] LIKE, IS [NOT] NULL, [NOT] IN	Pattern, null, and set comparison
7	[NOT] BETWEEN	Range comparison
8	!=,<>	Not equal to
9	NOT	NOT logical condition
10	AND	AND logical condition
11	OR	OR logical condition

When the NOT operator modifies the LIKE, IS NULL, and IN comparison operators, their precedence level remains the same as the positive form of these operators.

where last_name='King' where NOT (last_name='King')
where first_name LIKE 'R%' where first_name NOT LIKE 'R%'
where department_id IN (10,20,30) where department_id NOT IN (10,20,30)
where salary BETWEEN 1 and 3000 where salary NOT BETWEEN 1 and 3000
where commission_pct IS NULL where commission_pct IS NOT NULL

Consider the following SELECT statement that demonstrates the interaction of various different operators:

```
select last_name,salary,department_id,job_id,commission_pct
from employees
where last_name like '%a%' and salary > department_id * 200
or
job_id in ('MK_REP','MK_MAN') and commission_pct is not null
```

The LAST_NAME, SALARY, DEPARTMENT_ID, JOB_ID, and COMMISSION_PCT columns are projected from the EMPLOYEES table based on two discrete conditions. The first condition retrieves the records containing the character “a” in the LAST_NAME field AND with a SALARY value greater than 200 times the DEPARTMENT_ID value. The product of DEPARTMENT_ID and 200 is processed before the inequality operator since the precedence of multiplication is higher than inequality comparison. The second condition fetches those rows with JOB_ID values of either MK_MAN or MK_REP in which COMMISSION_PCT values are not null. For a row to be returned by this query, either the first OR second conditions need to be fulfilled.

Changing the order of the conditions in the WHERE clause changes its meaning due to the different precedence of the operators. Consider the following code sample:

```
select last_name,salary,department_id,job_id,commission_pct
from employees
where last_name like '%a%' and salary > department_id * 100 and
commission_pct is not null
or
job_id = 'MK_MAN'
```

There are two composite conditions in this query. The first condition retrieves the records with the character “a” in the LAST_NAME field AND a SALARY value greater than 100 times the DEPARTMENT_ID value AND where the COMMISSION_PCT value is not null. The second condition fetches those rows with JOB_ID values of MK_MAN. A row is returned by this query, if it conforms to either condition one OR condition two, but not necessarily to both.

Sort the Rows Retrieved by a Query

Regular language dictionaries sort words in alphabetical order. Pages printed in a book are sorted numerically in ascending order from beginning to end.

The usability of the retrieved datasets may be significantly enhanced with a mechanism to order or sort the information. Information may be sorted alphabetically, numerically, from earliest to latest, or in ascending or descending order. Further, the data may be sorted by one or more columns, including columns that are not listed in the SELECT clause. Sorting is performed once the results of a SELECT statement have been fetched. The sorting parameters do not influence the records returned by a query, just the presentation of the results. Exactly the same rows are returned by a statement including a sort clause as are returned by a statement excluding a sort clause. Only the ordering of the output may differ. Sorting the results of a query is accomplished using the ORDER BY clause.

The ORDER BY Clause

When tables are created, they are initially empty and contain no rows. As rows are inserted, updated, and deleted by one or more users or application systems, the original ordering of the stored rows is lost. The Oracle server cannot and does not guarantee that rows are stored sequentially. This is not a problem since a mechanism to sort the retrieved dataset is available in the form of the *ORDER BY* clause.

This clause is responsible for transforming the output of a query into more practical, user-friendly sorted data. The ORDER BY clause is always the last clause in a SELECT statement. As the full syntax of the SELECT statement is progressively exposed, you will observe new clauses added but none of them will be positioned after the ORDER BY clause. The format of the ORDER BY clause in the context of the SQL SELECT statement is as follows:

SELECT *|{[DISTINCT] *column|expression* [*alias*],…}
FROM *table*
[WHERE *condition(s)*]
[ORDER BY {*col(s)|expr|numeric_pos*} [ASC|DESC] [NULLS FIRST|LAST]];

Ascending and Descending Sorting

Ascending sort order is natural for most types of data and is therefore the default sort order used whenever the ORDER BY clause is specified. An ascending sort order for numbers is lowest to highest, while it is earliest to latest for dates and alphabetically for characters. The first form of the ORDER BY clause shows that results of a query may be sorted by one or more columns or expressions:

ORDER BY *col(s)|expr;*

Suppose that a report is requested that must contain an employee’s LAST_NAME, HIRE_DATE, and SALARY information, sorted alphabetically by the LAST_NAME column for all sales representatives and marketing managers. This report could be extracted with the following SELECT statement:

```
select last_name, hire_date, salary
from employees
where job_id in ('SA_REP','MK_MAN')
order by last_name;
```

The data selected may be ordered by any of the columns from the tables in the FROM clause, including those that do not appear in the SELECT list.

NULLS LAST keywords, which specify that if the sort column contains null values, then these rows are to be listed last after sorting the remaining rows based on their NOT NULL values. To specify that rows with null values in the sort column should be displayed first, append the *NULLS FIRST* keywords to the ORDER BY clause.

```
select last_name, salary, hire_date, hire_date-(salary/10) emp_value
from employees
where job_id in ('SA_REP','MK_MAN')
order by emp_value;
```

The EMP_VALUE expression is initialized with the HIRE_DATE value and is offset further into the past based on the SALARY field. The earliest EMP_VALUE date appears first in the result set output since the ORDER BY clause specifies that the results will be sorted by the expression alias. Note that the results could be sorted by the explicit expression and the alias could be omitted as in ORDER BY HIRE_DATE-(SALARY/10), but using aliases renders the query easier to read.

Several implicit default options are selected when you use the ORDER BY clause. The most important of these is that unless DESC is specified, the sort order is assumed to be ascending. If null values occur in the sort column, the default sort order is assumed to be NULLS LAST for ascending sorts and NULLS FIRST for descending sorts. If no ORDER BY clause is specified, the same query executed at different times may return the same set of results in different row order, so no assumptions should be made regarding the default row order.

Positional Sorting

Oracle offers an alternate and shorter way to specify the sort column or expression. Instead of specifying the column name, the position of the column as it occurs in the SELECT list is appended to the ORDER BY clause.

```
select last_name, hire_date, salary
from employees
where job_id in ('SA_REP','MK_MAN')
order by 2;
```

The ORDER BY clause specifies the numeric literal two. This is equivalent to specifying ORDER BY HIRE_DATE, since the HIRE_DATE column is the second column selected in the SELECT clause.

Positional sorting applies only to columns in the SELECT list that have a numeric position associated with them. Modifying the preceding query to sort the results by the JOB_ID column is not possible using positional sorting since this column does not occur in the SELECT list.

Composite Sorting

Results of a query may be sorted by more than one column using *composite sorting.* Two or more columns may be specified (either literally or positionally) as the composite sort key by commas separating them in the ORDER BY clause.

Consider the requirement to fetch the JOB_ID, LAST_NAME, SALARY, and HIRE_DATE values from the EMPLOYEES table. The further requirements are that the results must be sorted in reverse alphabetical order by JOB_ID first, then in ascending lphabetical order by LAST_NAME, and finally in numerically descending order based on the SALARY column.

The following SELECT statement fulfils these requirements:

```
select job_id, last_name, salary, hire_date
from employees
where job_id in ('SA_REP','MK_MAN')
order by job_id desc, last_name, 3 desc;
```

Each column involved in the sort is listed left to right in order of importance separated by commas in the ORDER BY clause, including the modifier DESC, which occurs twice in this clause. This example also demonstrates mixing literal and positional column specifications. As Figure 3-19 shows, there are several rows with the same JOB_ID value, for example, SA_REP. For these rows, the data is sorted alphabetically by the secondary sort key, which is the LAST_NAME column. For the rows with the same JOB_ID and same LAST_NAME column values such as SA_REP and Smith, these rows are sorted in numeric descending order by the third sort column, SALARY.

Ampersand Substitution

As queries are developed and perfected, they may be saved for future use. Sometimes, queries differ very slightly, and it is desirable to have a more generic form of the query that has a variable or placeholder defined that can be substituted at runtime.

Oracle offers this functionality in the form of *ampersand substitution.* Every element of the SELECT statement may be substituted, and the reduction of queries to their core elements to facilitate reuse can save you hours of tedious and repetitive work.

Substitution Variables

The key to understanding substitution variables is to regard them as placeholders. A SQL query is composed of two or more clauses. Each clause can be divided into subclauses, which are in turn made up of character text. Any text, subclause or clause element, or even the entire SQL query is a candidate for substitution.

Consider the SELECT statement in its general form:

SELECT *|{[DISTINCT] *column*|*expression* [*alias*],…}
FROM *table*
[WHERE *condition(s)*]
[ORDER BY {*col(s)*|*expr*|*numeric_pos*} [ASC|DESC] [NULLS FIRST|LAST]];

Using substitution, you insert values into the italicized elements, choosing which optional keywords to use in your queries. When the LAST_NAME column in the EMPLOYEES table is required, the query is constructed using the general form of the SELECT statement and substituting the column name: LAST_NAME in place of the word *column* in the SELECT clause and the table name; EMPLOYEES in place of the word *table* in the FROM clause.

Single Ampersand Substitution

The most basic and popular form of substitution of elements in a SQL statement is *single ampersand substitution.* The ampersand character (&) is the symbol chosen to designate a substitution variable in a statement and precedes the variable name with no spaces between them. When the statement is executed, the Oracle server processes the statement, notices a substitution variable, and attempts to resolve this variable's value in one of two ways.

First, it checks whether the variable is *defined* in the user session. If the variable is not defined, the user process prompts for a value that will be substituted in place of the variable. Once a value is submitted, the statement is complete and is executed by the Oracle server. The *ampersand substitution* variable is resolved at execution time and is sometimes known as *runtime binding* or *runtime substitution*.

A common requirement in the sample HR department may be to retrieve the same information for different employees at different times. Perhaps you are required to look up contact information like PHONE_NUMBER data given either LAST_NAME or EMPLOYEE_ID values. This generic request can be written as follows:

```
select employee_id, last_name, phone_number
from employees
where last_name = &LASTNAME
or employee_id = &EMPNO;
```

when running this query, Oracle server prompts you to input a value for the variable called LASTNAME. You enter an employee's last name, if you know it, for example, 'King'. If you don't know the last name but know the employee ID number, you can type in any value and press the ENTER key to submit the value. Oracle then prompts you to enter a value for the EMPNO variable. After typing in a value, for example, 0, and hitting ENTER, there are no remaining substitution variables for Oracle to resolve and the following statement is executed:

```
select employee_id, last_name, phone_number
from employees
where last_name = 'King'
or employee_id = 0;
```

Variables can be assigned any alphanumeric name that is a valid identifier name. The literal you substitute when prompted for a variable must be an appropriate data type for that context; otherwise, an ORA-00904: invalid identifier error is returned.

If the variable is meant to substitute a character or date value, the literal needs to be enclosed in single quotes. A useful technique is to enclose the *ampersand substitution* variable in single quotes when dealing with character and date values. In this way, the user is required to submit a literal value without worrying about enclosing it in quotes. The following statement rewrites the previous one but encloses the LASTNAME variable in quotes:

```
select employee_id, last_name, phone_number, email
from employees
where last_name = '&LASTNAME'
or employee_id = &EMPNO;
```

When prompted for a value to substitute for the LASTNAME variable, you may for example, submit the value King without any single quotes, as these are already present and when the *runtime substitution* is performed, the first WHERE clause condition will resolve to: WHERE LAST_NAME = 'King'.

Double Ampersand Substitution

There are occasions when a substitution variable is referenced multiple times in the same query. In such situations, the Oracle server will prompt you to enter a value for every occurrence of the single ampersand substitution variable.

For complex scripts this can be very inefficient and tedious. The following statement retrieves the FIRST_NAME and LAST_NAME columns from the EMPLOYEES table for those rows that contain the same set of characters in both these fields:

```
select first_name, last_name
from employees
where last_name like '%&SEARCH%'
and first_name like '%&SEARCH%';
```

The two conditions are identical but apply to different columns. When this statement is executed, you are first prompted to enter a substitution value for the SEARCH variable used in the comparison with the LAST_NAME column. Thereafter, you are prompted to enter a substitution value for the SEARCH variable used in the comparison with the FIRST_NAME column. This poses two problems. First, it is inefficient to enter the same value twice, but second and more importantly,
typographical errors may confound the query since Oracle does not verify that the same literal value is entered each time substitution variables with the same name are used. In this example, the logical assumption is that the contents of the variables substituted should be the same, but the fact that the variables have the same name has no meaning to the Oracle server and it makes no such assumption.

The first example shows the results of running the preceding query and submitting two distinct values for the SEARCH substitution variable. In this particular example, the results are incorrect since the requirement was to retrieve FIRST_NAME and LAST_NAME pairs which contained the identical string of characters.

In situations when a substitution variable is referenced multiple times in the same query and your intention is that the variable must have the same value at each occurrence in the statement, it is preferable to make use of *double ampersand substitution*. This involves prefixing the first occurrence of the substitution variable that occurs multiple times in a query, with two ampersand symbols instead of one. When the Oracle server encounters a *double ampersand substitution* variable, a session value is defined for that variable and you are not prompted to enter a value to be substituted for this variable in subsequent references.

The second example demonstrates how the SEARCH variable is preceded by two ampersands in the condition with the FIRST_NAME column and thereafter is prefixed by one ampersand in the condition with the LAST_NAME column. When executed, you are prompted to enter a value to be substituted for the SEARCH variable only once for the condition with the FIRST_NAME column. This value is then automatically resolved from the session value of the variable in subsequent references to it, as in the condition with the LAST_NAME column. To undefine the SEARCH variable, you need to use the *UNDEFINE* command.

Substituting Column Names

Literal elements of the WHERE clause have been the focus of the discussion on substitution thus far, but virtually any element of a SQL statement is a candidate for substitution. In the following statement, the FIRST_NAME and JOB_ID columns are static and will always be retrieved, but the third column selected is variable and specified as a substitution variable named: COL. The result set is further sorted by this variable column in the ORDER BY clause:

```
select first_name, job_id, &&col
from employees
where job_id in ('MK_MAN','SA_MAN')
order by &col;
```

At runtime, you are prompted to provide a value for the double ampersand variable called COL. You could, for example, enter the column named SALARY and submit your input. The statement that executes performs the substitution and retrieves the FIRST_NAME, JOB_ID, and SALARY columns from the EMPLOYEES table sorted by SALARY.

Unlike character and date literals, column name references do not require single quotes both when explicitly specified and when substituted via ampersand substitution.

Substituting Expressions and Text

Almost any element of a SQL statement may be substituted at runtime. The constraint is that Oracle requires at least the first word to be static. In the case of the SELECT statement, at the very minimum, the SELECT keyword is required and the remainder of the statement may be substituted as follows:

```
select &rest_of_statement;
```

When executed, you are prompted to submit a value for the variable called: REST_OF_STATEMENT, which could be any legitimate query, such as DEPARTMENT_NAME from DEPARMENTS. If you submit this text as input for the variable, the query that is run will be resolved to the following statement:

```
select department_name from departments;
```

Consider the general form of the SQL statement rewritten using ampersand substitution, as shown next:

```
select &SELECT_CLAUSE
from &FROM_CLAUSE
where &WHERE_CLAUSE
order by &ORDER_BY_CLAUSE;
```

The preceding statement allows any query discussed so far to be submitted at runtime. The first execution queries the REGIONS table, while the second execution queries the COUNTRIES table. Useful candidates for ampersand substitution are statements that are run multiple times and differ slightly from each other.

Define and Verify

Double ampersand substitution is used to avoid repetitive input when the same variable occurs multiple times in a statement. When a double ampersand substitution occurs, the variable is stored as a session variable. As the statement executes, all further occurrences of the variable are automatically resolved using the stored session variable. Any subsequent executions of the statement within the same session automatically resolve the substitution variables from stored session values.
This is not always desirable and indeed limits the usefulness of substitution variables. Oracle does, however, provide a mechanism to *UNDEFINE* these session variables.

The *VERIFY* command is specific to SQL*Plus and controls whether or not substituted elements are echoed on the user's screen prior to executing a SQL statement that uses substitution variables.

The DEFINE and UNDEFINE Commands

Session level variables are implicitly created when they are initially referenced in SQL statements using double ampersand substitution. They persist or remain available for the duration of the session or until they are explicitly undefined. A session ends when the user exits their client tool like SQL*Plus or when the user process is terminated abnormally.

The problem with persistent session variables is they tend to detract from the generic nature of statements that use ampersand substitution variables. Fortunately, these session variables can be removed with the UNDEFINE command. Within a script or at the command line of SQL*Plus or SQL Developer, the syntax for undefining session variables is as follows:
UNDEFINE *variable*;
Consider a simple generic example that selects a static and variable column from the EMPLOYEES table and sorts the output based on the variable column. The static column could be the LAST_NAME column.

```
select last_name, &&COLNAME
from employees
where department_id=30
order by &COLNAME;
```

The first time this statement executes, you are prompted to input a value for the variable called COLNAME. Let's assume you enter SALARY. This value is substituted and the statement executes. A subsequent execution of this statement within the same session does not prompt for any COLNAME values since it is already defined as SALARY in the context of this session and can only be undefined with the UNDEFINE COLNAME command. Once the variable has been undefined, the next execution of the statement prompts the user for a value for the COLNAME variable.

The DEFINE command serves two purposes. It can be used to retrieve a list of all the variables currently defined in your SQL session; it can also be used to explicitly define a value for a variable referenced as a substitution variable by one or more statements during the lifetime of that session. The syntax for the two variants of the DEFINE command are as follows:
DEFINE;
DEFINE *variable=value;*
A variable called EMPNAME is defined explicitly to have the value 'King'. The stand-alone DEFINE command in SQL*Plus then returns a number of session variables prefixed with an underscore character as well as other familiar variables, including EMPNAME and double ampersand substitution variables implicitly defined earlier. Two different but simplistic query examples are executed, and the explicitly defined substitution variable EMPNAME is referenced by both queries. Finally, the variable is UNDEFINED.
The capacity of the SQL client tool to support session-persistent variables may be switched off and on as required using the *SET* command. The SET command is not a SQL language command, but rather a SQL environment control command. By specifying SET DEFINE OFF, the client tool (for example, SQL*Plus) does not save session variables or attach special meaning to the ampersand symbol. This allows the ampersand symbol to be used as an ordinary literal character if necessary. The SET DEFINE ON|OFF command therefore determines whether or not ampersand substitution is available in your session.

The following example uses the ampersand symbol as a literal value. When executed, you are prompted to submit a value for bind variable SID.

```
select 'Coda & Sid' from dual;
```

By turning off the ampersand substitution functionality as follows, this query may be executed without prompts:

```
SET DEFINE OFF
select 'Coda & Sid' from dual;
SET DEFINE ON
```

Once the statement executes, the SET DEFINE ON command may be used to switch the substitution functionality back on. If DEFINE is SET OFF and the context that an ampersand is used in a statement cannot be resolved literally, Oracle returns an error.

The VERIFY Command

As discussed earlier, two categories of commands are available when dealing with the Oracle server: SQL language commands and the SQL client control commands. The SELECT statement is an example of a language command, whilst the SET command controls the SQL client environment. There are many different language and control commands available, but the control commands pertinent to substitution are DEFINE and *VERIFY.*

The VERIFY command controls whether the substitution variable submitted is displayed onscreen so you can *verify* that the correct substitution has occurred. A message is displayed showing the *old* clause followed by the *new* clause containing the substituted value. The VERIFY command is switched ON and OFF with the command SET VERIFY ON|OFF. VERIFY is first switched OFF, a query that uses ampersand substitution is executed, and you are prompted to input a value.
The value is then substituted, the statement runs, and its results are displayed. VERIFY is then switched ON, the same query is executed, and you are prompted to input a value. Once the value is input and before the statement commences

execution, Oracle displays the clause containing the reference to the substitution variable as the *old* clause with its line number and, immediately below this, the *new* clause displays the statement containing the substituted value.

✓ TWO-MINUTE DRILL

Limit the Rows Retrieved by a Query

❑ The WHERE clause extends the SELECT statement by providing the language that enables selection.

❑ One or more conditions constitute a WHERE clause. These conditions specify rules to which the data in a row must conform to be eligible for selection.

❑ For each row tested in a condition, there are terms on the left and right of a comparison operator. Terms in a condition can be column values, literals, or expressions.

❑ Comparison operators may test two terms in many ways. Equality or inequality tests are very common, but range, set, and pattern comparisons are also available.

❑ Range comparison is performed using the BETWEEN operator, which tests whether a term falls between given start and end boundary values.

❑ Set membership is tested using the IN operator. A condition based on a set comparison evaluates to true if the left-side term is listed in the single-quoted, comma-delimited set on the right-side.

❑ The LIKE operator enables literal character patterns to be matched with other literals, column values, or evaluated expressions. The percentage symbol (%) behaves as a wildcard that matches zero or more characters. The underscore symbol (_) behaves as a single character wildcard that matches exactly one other character.

❑ Boolean operators include the AND, OR, and NOT operators. The AND and OR operators enable multiple conditional clauses to be specified. These are sometimes referred to as multiple WHERE clauses.

❑ The NOT operator negates the comparison operator involved in a condition.

❑ Results are sorted using the ORDER BY clause. Rows retrieved may be ordered according to one or more columns by specifying either the column names or their numeric position in the SELECT clause.

❑ The sorted output may be arranged in descending or ascending order using the DESC or ASC modifiers after the sort terms in the ORDER BY clause.

❑ Ampersand substitution facilitates SQL statement reuse by providing a means to substitute elements of a statement at runtime. The same SQL statement may therefore be run multiple times with different input parameters.

❑ Single ampersand substitution requires user input for every occurrence of the substitution variable in the statement. Double ampersand substitution requires user input only once per occurrence of a substitution variable, since it defines a session-persistent variable with the given input value.

❑ Session-persistent variables may be set explicitly using the DEFINE command. The UNDEFINE command may be used to unset both implicitly (double ampersand substitution) and explicitly defined session variables.

❑ The VERIFY environmental setting controls whether SQL*Plus displays the old and new versions of statement lines which contain substitution variables.

SELF TEST

Read all the choices carefully because there may be more than one correct answer. Choose all the correct answers for each question.

1. Which two clauses of the SELECT statement facilitate selection and projection?
A. SELECT, FROM
B. ORDER BY, WHERE
C. SELECT, WHERE
D. SELECT, ORDER BY

2. Choose the query that extracts the LAST_NAME, JOB_ID, and SALARY values from the EMPLOYEES table for records having JOB_ID values of either SA_REP or MK_MAN and having SALARY values in the range of $1000 to $4000. The SELECT and FROM clauses are SELECT LAST_NAME, JOB_ID, SALARY FROM EMPLOYEES:
A. WHERE JOB_ID IN ('SA_REP','MK_MAN') AND SALARY > 1000 AND SALARY < 4000;
B. WHERE JOB_ID IN ('SA_REP','MK_MAN') AND SALARY BETWEEN 1000 AND 4000;
C. WHERE JOB_ID LIKE 'SA_REP%' AND 'MK_MAN%' AND SALARY > 1000 AND SALARY < 4000;
D. WHERE JOB_ID = 'SA_REP' AND SALARY BETWEEN 1000 AND 4000 OR JOB_ID='MK_MAN';

3. Which of the following WHERE clauses contains an error? The SELECT and FROM clauses are
SELECT * FROM EMPLOYEES:
A. WHERE HIRE_DATE IN ('02-JUN-2004');
B. WHERE SALARY IN ('1000','4000','2000');
C. WHERE JOB_ID IN (SA_REP,MK_MAN);
D. WHERE COMMISSION_PCT BETWEEN 0.1 AND 0.5;

4. Choose the WHERE clause that extracts the DEPARTMENT_NAME values containing the character literal "er" from the DEPARTMENTS table. The SELECT and FROM clauses are
SELECT DEPARTMENT_NAME FROM DEPARTMENTS:
A. WHERE DEPARTMENT_NAME IN ('%e%r');
B. WHERE DEPARTMENT_NAME LIKE '%er%';
C. WHERE DEPARTMENT_NAME BETWEEN 'e' AND 'r';
D. WHERE DEPARTMENT_NAME CONTAINS 'e%r';

5. Which two of the following conditions are equivalent to each other?
A. WHERE COMMISSION_PCT IS NULL
B. WHERE COMMISSION_PCT = NULL
C. WHERE COMMISSION_PCT IN (NULL)
D. WHERE NOT(COMMISSION_PCT IS NOT NULL)

6. Which three of the following conditions are equivalent to each other?
A. WHERE SALARY <=5000 AND SALARY >=2000
B. WHERE SALARY IN (2000,3000,4000,5000)
C. WHERE SALARY BETWEEN 2000 AND 5000
D. WHERE SALARY > 1999 AND SALARY < 5001
E. WHERE SALARY >=2000 AND <=5000

7. Choose one false statement about the ORDER BY clause.
A. When using the ORDER BY clause, it always appears as the last clause in a SELECT statement.

B. The ORDER BY clause may appear in a SELECT statement that does not contain a WHERE clause.
C. The ORDER BY clause specifies one or more terms by which the retrieved rows are sorted. These terms can only be column names.
D. Positional sorting is accomplished by specifying the numeric position of a column as it appears in the SELECT list, in the ORDER BY clause.

8. The following query retrieves the LAST_NAME, SALARY, and COMMISSION_PCT values for employees whose LAST_NAME begins with the letter R. Based on the following query, choose the ORDER BY clause that first sorts the results by the COMMISSION_PCT column, listing highest commission earners first, and then sorts the results in ascending order by the SALARY column. Any records with NULL COMMISSION_PCT must appear last:

```
SELECT LAST_NAME, SALARY, COMMISSION_PCT
FROM EMPLOYEES
WHERE LAST_NAME LIKE 'R%'
```

A. ORDER BY COMMISSION_PCT DESC, 2;
B. ORDER BY 3 DESC, 2 ASC NULLS LAST;
C. ORDER BY 3 DESC NULLS LAST, 2 ASC;
D. ORDER BY COMMISSION_PCT DESC, SALARY ASC;

9. The DEFINE command explicitly declares a session-persistent substitution variable with a specific value. How is this variable referenced in an SQL statement? Consider an expression that calculates tax on an employee's SALARY based on the current tax rate. For the following session-persistent substitution variable, which statement correctly references the TAX_RATE variable?

```
DEFINE TAX_RATE=0.14
```

A. SELECT SALARY * :TAX_RATE TAX FROM EMPLOYEES;
B. SELECT SALARY * &TAX_RATE TAX FROM EMPLOYEES;
C. SELECT SALARY * :&&TAX TAX FROM EMPLOYEES;
D. SELECT SALARY * TAX_RATE TAX FROM EMPLOYEES;

10. When using ampersand substitution variables in the following query, how many times will you be prompted to input a value for the variable called JOB the first time this query is executed?

```
SELECT FIRST_NAME, '&JOB'
FROM EMPLOYEES
WHERE JOB_ID LIKE '%'||&JOB||'%'
AND '&&JOB' BETWEEN 'A' AND 'Z';
```

A. 0
B. 1
C. 2
D. 3

LAB QUESTION

Using SQL Developer or SQL*Plus, connect to the OE schema and complete the following tasks. A customer requires a hard disk drive and a graphics card for her personal computer. She is willing to spend between $500 and $800 on the disk drive but is unsure about the cost of a graphics card. Her only requirement is that the resolution supported by the graphics card should be either 1024·768 or 1280·1024. As the sales representative, you have been tasked to write one query that searches the PRODUCT_INFORMATION table where the PRODUCT_NAME value begins with HD (hard disk) or GP (graphics processor) and their list prices. Remember the hard disk list prices must be between $500 and $800 and the graphics processors need to support either 1024·768 or 1280·1024. Sort the results in descending LIST_PRICE order.

SELF TEST ANSWERS

Limit the Rows Retrieved by a Query

1. ✹✓ **C.** The SELECT clause facilitates projection by specifying the list of columns to be projected from a table, whilst the WHERE clause facilitates selection by limiting the rows retrieved based on its conditions.
✹⋎ **A**, **B**, and **D** are incorrect because the FROM clause specifies the source of the rows being projected and the ORDER BY clause is used for sorting the selected rows.

2. ✹✓ **B.** The IN operator efficiently tests whether the JOB_ID for a particular row is either SA_REP or MK_MAN, whilst the BETWEEN operator efficiently measures whether an employee's SALARY value falls within the required range.
✹⋎ **A** and **C** exclude employees who earn a salary of $1000 or $4000, since these SALARY values are excluded by the inequality operators. C also selects JOB_ID values like SA_REP% and MK_MAN%, potentially selecting incorrect JOB_ID values. **D** is half right. The first half returns the rows with JOB_ID equal to SA_REP having SALARY values between $1000 and $4000. However, the second part (the OR clause), correctly tests for JOB_ID equal to MK_MAN but ignores the SALARY condition.

3. ✹✓ **C.** The character literals being compared to the JOB_ID column by the IN operator must be enclosed by single quotation marks.
✹⋎ **A**, **B**, and **D** are syntactically correct. Notice that B does not require quotes around the numeric literals. Having them, however, does not cause an error.

4. ✹✓ **B.** The LIKE operator tests the DEPARTMENT_NAME column of each row for values that contain the characters "er". The percentage symbols before and after the character literal indicate that any characters enclosing the "er" literal are permissible.
✹⋎ **A** and **C** are syntactically correct. **A** uses the IN operator, which is used to test set membership. **C** tests whether the alphabetic value of the DEPARTMENT_NAME column is between the letter "e" and the letter "r." Finally, **D** uses the word "contains," which cannot be used in this context.

5. ✹✓ **A** and **D.** The IS NULL operator correctly evaluates the COMMISSION_PCT column for NULL values. **D** uses the NOT operator to negate the already negative version of the IS NULL operator, IS NOT NULL. Two negatives return a positive, and therefore **A** and **D** are equivalent.
✹⋎ **B** and **C** are incorrect since NULL values cannot be tested by the equality operator or the IN operator.

6. ✹✓ **A, C,** and **D.** Each of these conditions tests for SALARY values in the range of $2000 to $5000.
✹⋎ **B** and **E** are incorrect. **B** excludes values like $2500 from its set, and **E** is illegal since it is missing the SALARY column name reference after the AND operator.

7. ✹✓ **C.** The terms specified in an ORDER BY clause can include column names, positional sorting, numeric values, and expressions.
✹⋎ **A, B,** and **D** are true.

8. ✹✓ **C.** Positional sorting is performed, and the third term in the SELECT list, COMMISSION_PCT, is sorted first in descending order, and any NULL COMMISSION_PCT values are listed last. The second term in the SELECT list, SALARY, is sorted next in ascending order.
✹⋎ **A, B,** and **D** are incorrect. **A** does not specify what to do with NULL COMMISSION_PCT values, and the default behavior during a descending sort is to list NULLS FIRST. **B** applies the NULLS LAST modifier to the SALARY column instead of the COMMISSION_PCT column, and **D** ignores NULLS completely.

9. ✹✓ **B.** A session-persistent substitution variable may be referenced using an ampersand symbol from within any SQL statement executed in that session.
✹⮟ **A, C,** and **D** are incorrect. **A** and **D** attempt to reference the substitution variable using a colon prefix to its name and the variable name on its own. These are invalid references to substitution variables in SQL. **C** references a variable called TAX and not the variable TAX_RATE.

10. ✹✓ **D.** The first time this statement is executed, two single ampersand substitution variables are encountered before the third double ampersand substitution variable. If the first reference on line one of the query contained a double ampersand substitution, you would only be prompted to input a value once.
✹⮟ **A, B,** and **C** are incorrect since you are prompted thrice to input a value for the JOB substitution variable. In subsequent executions of this statement in the same session you will not be prompted to input a value for this variable.

LAB ANSWER

Using SQL Developer or SQL*Plus, connect to the OE schema and complete the following tasks. You are required to query the PRODUCT_INFORMATION table in the OE schema for the PRODUCT_NAME and LIST_PRICE columns. The rows selected must conform to either of two conditions. The first condition is that the PRODUCT_NAME must begin with the characters 'HD' and their LIST_PRICE must fall in the range between $500 and $800. Alternately, the row can conform to the second condition that the PRODUCT_NAME must begin with the characters 'GP' and contain the characters '1024'. Finally, the results must be sorted in descending LIST_PRICE order.

```
SELECT PRODUCT_NAME, LIST_PRICE
FROM PRODUCT_INFORMATION
WHERE (PRODUCT_NAME LIKE 'HD%' AND LIST_PRICE BETWEEN 500 AND 800)
OR (PRODUCT_NAME LIKE 'GP%1024%')
ORDER BY LIST_PRICE DESC
```

Chapter 3: Single-Row Functions

Oracle server implements a proprietary procedural language called PL/SQL, or procedural SQL. A range of named programmatic objects may be constructed using PL/SQL. These include procedures, functions, and packages.
The functions discussed here are confined to PL/SQL programs packaged and supplied by Oracle as built-in features.

Defining a Function

A *function* is a program written to optionally accept input parameters, perform an operation, or return a single value. A function returns only one value per execution.

Three important components form the basis of defining a function.

The first is the input parameter list. It specifies zero or more arguments that may be passed to a function as input for processing. These arguments or parameters may be of differing data types, and some are mandatory while others may be optional.

The second component is the data type of its resultant value. Upon execution, only one value is returned by the function.

The third encapsulates the details of the processing performed by the function and contains the program code that optionally manipulates the input parameters, performs calculations and operations, and generates a return value.

A function is often described as a *black box* that takes an input, performs a calculation, and returns a value as illustrated by the following equation. Instead of focusing on their implementation details, you are encouraged to concentrate on the features that built-in functions provide.
F(x, y, z, …) = result;
Functions may be *nested* within other functions, such as F1(x, y, F2(a, b), z), where F2, which takes two input parameters, a and b, and forms the third of four parameters submitted to F1. Functions can operate on any available data types, the most popular being character, date, and numeric data. These operands may be columns or expressions.

Operating on Character Data

Character data or strings are versatile since they facilitate the storage of almost any type of data.
Functions that operate on character data are broadly classified as *case conversion* and *character manipulation* functions.
LOWER, UPPER, and INITCAP are the case conversion functions that convert a given character column, literal, or expression into lowercase, uppercase, or initial case, respectively:

```
lower('SQL') = sql
upper('sql') = SQL
initcap('sql') = Sql
```

The character manipulation functions are exceptionally powerful and include the LENGTH, CONCAT, SUBSTR, INSTR, LPAD, RPAD, TRIM, and REPLACE functions.

The LENGTH(*string*) function uses a character string as an input parameter and returns a numeric value representing the number of characters present in that string:

```
length('A short string') = 14
```

The CONCAT(*string 1, string 2*) function takes two strings and concatenates or joins them in the same way that the concatenation operator || does:

```
concat('SQL is',' easy to learn.') = SQL is easy to learn.
```

The SUBSTR(*string, start position, number of characters*) function accepts three parameters and returns a string consisting of the number of characters extracted from the source string, beginning at the specified start position:

```
substr('http://www.domain.com',12,6) = domain
```

The INSTR(source string, search item, [start position],[*nth* occurrence of search item]) function returns a number that represents the position in the source string, beginning from the given start position, where the *nth* occurrence of the search item begins:

```
instr('http://www.domain.com','.',1,2) = 18
```

The LPAD(*string, length after padding, padding string*) and RPAD(*string, length after padding, padding string*) functions add a padding string of characters to the left or right of a string until it reaches the specified length after padding. The TRIM function literally trims off leading or trailing (or both) character strings from a given source string:

```
rpad('#PASSWORD#',11,'#') = #PASSWORD##
lpad('#PASSWORD#',11,'#') = ##PASSWORD#
trim('#' from '#PASSWORD#') = PASSWORD
```

The REPLACE(*string, search item, replacement item*) function locates the search item in a given string and replaces it with the replacement item, returning a string with replaced values:

```
replace('#PASSWORD#','WORD','PORT') = #PASSPORT#
```

Operating on Numeric Data

Many numeric built-in functions are available. Some calculate square roots, perform exponentiation, and convert numbers into hexadecimal format. There are too many to mention, and many popular mathematical, scientific, and financial calculations have been exposed as built-in functions by Oracle.
Three common numeric functions, are ROUND, TRUNC, and MOD.
ROUND(*number, decimal precision*) facilitates rounding off a number to the lowest or highest value given a decimal precision format:

```
round(42.39,1) = 42.4
```

The TRUNC(*number, decimal precision*) function drops off or truncates the number given a decimal precision value:

```
trunc(42.39,1) = 42.3
```

The MOD(*dividend, divisor*) returns the remainder of a division operation:

```
mod(42,10) = 2
```

Operating on Date Information

Working with date values may be challenging. Performing date arithmetic that accommodates leap years and variable month lengths can be frustrating and error prone. Oracle addresses this challenge by providing native support for date
arithmetic and several built-in date functions such as MONTHS_BETWEEN, ADD_MONTHS, LAST_DAY, NEXT_DAY, SYSDATE, ROUND, and TRUNC.
The MONTHS_BETWEEN(*date 1, date 2*) function returns the number of months between two dates, while the ADD_MONTHS(*date 1, number of months*) returns the date resulting from adding a specified number of months to a date:

```
months_between('01-FEB-2008','01-JAN-2008') = 1
add_months('01-JAN-2008',1) = 01-FEB-2008
```

The LAST_DAY(*date 1*) function returns the last day of the month that the specified date falls into, while the NEXT_DAY(*date 1, day of the week*) returns the date on which the next specified day of the week falls after the given date:

```
last_day('01-FEB-2008') = 29-FEB-2008
next_day('01-FEB-2008','Friday') = 08-FEB-2008
```

The SYSDATE function takes no parameters and returns a date value that represents the current server date and time. ROUND(*date, date precision format*) and TRUNC(*date, date precision format*) round and truncate a given date value to the nearest date precision format like day, month, or year:

```
sysdate = 17-DEC-2007
round(sysdate,'month') = 01-JAN-2008
trunc(sysdate,'month') = 01-DEC-2007
```

Types of Functions

Two broad types of functions operating on single and multiple rows.

Single-Row Functions

There are several categories of *single-row* functions including character, numeric, date, conversion, and general. These are functions that operate on one row of a dataset at a time. If a query selects 10 rows, the function is executed 10 times, once per row with the values from that row as input to the function.

Single-row functions manipulate the data items in a row to extract and format them for display purposes. The input values to a single-row function can be userspecified constants or literals, column data, variables, or expressions optionally supplied by other nested single-row functions. The nesting of single-row functions is a commonly used technique. Functions can return a value with a different data type from its input parameters. The LENGTH function accepts one character input parameter and returns a numeric output.

They simplify working with NULL values and facilitate conditional logic within a SELECT statement. These include the NVL, NVL2, NULLIF, COALESCE, CASE, and DECODE functions.

Apart from their inclusion in the SELECT list of a SQL query, single-row functions may be used in the WHERE and ORDER BY clauses.

Multiple-Row Functions

As the name suggests, this category of functions operates on more than one row at a time. Typical uses of *multiple-row* functions include calculating the sum or average of the numeric column values or counting the total number of records in sets. These are sometimes known as *aggregation* or *group* functions

Using Character Case Conversion Functions

The character *case conversion* functions serve two important purposes. They may be used first, to modify the appearance of a character data item for display purposes and second, to render them consistent for comparison operations. It is simpler to search for a string using a consistent case format instead of testing every permutation of uppercase and lowercase characters that could match the string. It is important to remember that these functions do not alter the data stored in tables. They still form part of the read-only SQL query.

The character functions expect string parameters. These may be any string literal, character column value, or expression resulting in a character value. If it is a numeric or a date value, it is implicitly converted into a string.

The LOWER Function

The LOWER function converts a string of characters into their lowercase equivalents. Numeric, punctuation, or special characters are ignored.

The LOWER function can take only one parameter. Its syntax is LOWER(*s*). The following queries illustrate the usage of this function:

```
Query 1: select lower(100) from dual
Query 2: select lower(100+100) from dual
Query 3: select lower('The SUM '||'100+100'||' = 200') from dual
```

Queries 1 and 2 return the strings 100 and 200, respectively. The parameter to the LOWER function in query 3 is a character expression and the string returned by the function is “the sum 100 + 100 = 200.”

```
Query 4: select lower(SYSDATE) from dual
Query 5: select lower(SYSDATE+2) from dual
```

Assume that the current system date is: 17-DEC-2019. Queries 4 and 5 return the strings 17-dec-2019 and 19-dec-2019, respectively. The date expressions are evaluated and implicitly converted into character data before the LOWER function is executed.

```
select first_name, last_name
from employees
where lower(last_name) like '%ur%'
```

Consider writing an alternative query to return the same results without using the LOWER function. It could be done as follows:

```
select first_name, last_name
from employees
where last_name like '%ur%'

or last_name like '%UR%'
or last_name like '%uR%'
or last_name like '%Ur%'
```

This query works but is cumbersome, and the number of OR clauses required increases exponentially as the length of the search string increases.

The UPPER Function

All lowercase characters are converted into their uppercase equivalents. Numeric, punctuation, or special characters are ignored.

The UPPER function takes only one parameter. Its syntax is UPPER(*s*). The following queries illustrate the usage of this function:

```
Query 1: select upper(1+2.14) from dual
Query 2: select upper(SYSDATE) from dual
```

Query 1 returns the string 3.14. The parameter to the UPPER function in query 2 is SYSDATE, which returns the current system date. Since this value is returned in uppercase by default, no case conversion is performed.

The INITCAP Function

The INITCAP function converts a string of characters into capitalized case. It is often used for data presentation purposes. The first letters of each word in the string are converted to their uppercase equivalents, while the remaining letters of each word are converted to their lowercase equivalents. A word is usually a string of adjacent characters separated by a space or underscore, but other characters such as the percentage symbol, exclamation mark, or dollar sign are valid word separators. Punctuation or special characters are regarded as valid word separators.

The INITCAP function can take only one parameter. Its syntax is INITCAP(*s*).

The following queries illustrate the usage of this function:

```
Query 1: select initcap(21/7) from dual
Query 2: select initcap(SYSDATE) from dual
Query 3: select initcap('init cap or init_cap or init%cap') from dual
```

Query 1 returns 3.

Query 2 Assuming that the current system date is 17-DEC-2007, query 2 therefore returns 17-Dec-2007.

Query 3 returns Init Cap Or Init_Cap Or Init%Cap.

Using Character Manipulations Functions

Nesting these functions is common. The concatenation operator (||) is generally used instead of the CONCAT function. The LENGTH, INSTR, SUBSTR, and REPLACE functions often find themselves in each other's company as do RPAD, LPAD, and TRIM.

The CONCAT Function

The CONCAT function joins two character literals, columns, or expressions to yield one larger character expression. Numeric and date literals are implicitly cast as characters when they occur as parameters to the CONCAT function. Numeric or date expressions are evaluated before being converted to strings ready to be concatenated. The CONCAT function takes two parameters. Its syntax is CONCAT(*s1, s2*), where *s1* and *s2* represent string literals, character column values, or expressions resulting in character values. The following queries illustrate the usage of this function:

```
Query 1: select concat(1+2.14,' approximates pi') from dual
Query 2: select concat('Today is:',SYSDATE) from dual
```

Query 1 returns the string "3.14 approximates pi."

Query 2 If the system date is 17-DEC-2007, query 2 returns the string "Today is 17-DEC-2007."

```
select concat('Outer1 ', concat('Inner1',' Inner2')) from dual;
```

This query results in the following string: Outer1 Inner1 Inner2.

The LENGTH Function

The LENGTH function returns the number of characters that constitute a character string. This includes character literals, columns, or expressions. Numeric and date literals are automatically cast as characters when they occur as parameters to the LENGTH function. Numeric or date expressions are evaluated before being converted to strings ready to be measured. Blank spaces, tabs, and special characters are all counted by the LENGTH function.

The LENGTH function takes only one parameter. Its syntax is LENGTH(*s*), where *s* represents any string literal, character column value, or expression resulting in a character value.

```
Query 1: select length(1+2.14||' approximates pi') from dual
Query 2: select length(SYSDATE) from dual
```

Query 1 returns the number 20.

Query 2 Assuming that the system date returned is 17-DEC-07, query 2 returns value 9.

The LPAD and RPAD Functions

The LPAD and RPAD functions, also known as left pad and right pad functions, return a string padded with a specified number of characters to the left or right of the source string respectively. The character strings used for padding include character literals, column values, or expressions. Numeric and date literals are implicitly cast as characters when they occur as parameters to the LPAD or RPAD functions. Numeric or date expressions are evaluated before being converted to strings destined for padding. Blank spaces, tabs, and special characters may be used as padding characters.

The LPAD and RPAD functions take three parameters. Their syntaxes are LPAD(*s, n, p*) and RPAD(*s, n, p*), where *s* represents the source string, *n* represents the final length of the string returned, and *p* specifies the character string to be used as padding. If LPAD is used, the padding characters *p* are added to the left of the source string *s* until it reaches length *n*. If RPAD is used, the padding characters *p* are added to the right of the source string *s* until it reaches length *n*. Note that if the parameter *n* is smaller than or equal to the length of the source string *s*, then no padding occurs and only the first *n* characters of *s* are returned.

The following queries illustrate the usage of this function:

```
Query 1: select lpad(1000+200.55,14,'*') from dual
Query 2: select rpad(1000+200.55,14,'*') from dual
Query 3: select lpad(SYSDATE,14,'$#') from dual
Query 4: select rpad(SYSDATE,4,'$#') from dual
```

Query 1 returns a 14-character string: *******1200.55.

Query 2 returns the string "1200.55*******."

Query 3 returns: $#$#$17-DEC-07.

Query 4 The RPAD function in has a target length of 4 characters, but the SYSDATE function alone returns a 9-character value. Therefore no padding occurs and, assuming the current system date is 17-DEC-07, the first four characters of the converted date are returned: 17-D.

The TRIM Function

The TRIM function removes characters from the beginning or end of character literals, columns or expressions to yield one potentially shorter character item. Numeric and date literals are automatically cast as characters when they occur as parameters to the TRIM function. Numeric or date expressions are evaluated first before being converted to strings ready to be trimmed.

The TRIM function takes a parameter made up of an optional and a mandatory component. Its syntax is TRIM([*trailing*|*leading*|*both*] *trimstring* from *s*).

The string to be trimmed (*s*) is mandatory. The following points list the rules governing the use of this function:

- TRIM(*s*) removes spaces from both sides of the input string.
- TRIM(*trailing trimstring* from *s*) removes all occurrences of *trimstring* from the end of the string *s* if it is present.
- TRIM(*leading trimstring* from *s*) removes all occurrences of *trimstring* from the beginning of the string *s* if it is present.
- TRIM(*both trimstring* from *s*) removes all occurrences of *trimstring* from the beginning and end of the string *s* if it is present.

```
Query 1: select trim(trailing 'e' from 1+2.14||' is pie') from dual
Query 2: select trim(both '*' from '*******Hidden*******') from dual
Query 3: select trim(1 from sysdate) from dual
```

Query 1 return "3.14 approximates pi."

Query 2 returns the string "Hidden."

Query 3 Since no keyword is specified for trailing, leading, or both trim directions, the default of both applies. Therefore all occurrences of character 1 at the beginning or ending of the date string are trimmed resulting in 7-DEC-07 being returned.

Remember that when no parameters other than the string *s* are specified to the TRIM function then its default behavior is to trim(both ' ' from *s*).

The INSTR Function (In-string)

The INSTR function locates the position of a search string within a given string. It returns the numeric position at which the *nth* occurrence of the search string begins, relative to a specified start position. If the search string is not present the INSTR function returns zero.

Numeric and date literals are implicitly cast as characters when they occur as parameters to the INSTR function. Numeric or date expressions are first evaluated before being converted to strings ready to be searched.

The INSTR function takes four parameters made up of two optional and two mandatory arguments. The syntax is INSTR(*source string*, *search string*, [*search start position*], [*nth occurrence*]). The default value for the *search start position* is 1 or the beginning of the *source string*. The default value for the *nth* occurrence is 1 or the first occurrence.

```
Query 1: select instr(3+0.14,'.') from dual
Query 2: select instr(sysdate, 'DEC') from dual
Query 3: select instr('1#3#5#7#9#','#') from dual
Query 4: select instr('1#3#5#7#9#','#',5) from dual
Query 5: select instr('1#3#5#7#9#','#',3,4) from dual
```

Query 1 The period character is searched for and the first occurrence of it occurs at position 2.
Query 2 Assume that the current system date is 17-DEC-07. The first occurrence of the characters DEC occurs at position 4.
Query 3 returns 2.
Query 4 has the number 5 as its third parameter indicating that the search for the hash symbol must begin at position 5 in the source string. The subsequent occurrence of the hash symbol is at position 6, which is returned by the query. Query 5 has the numbers 3 and 4 as its third and fourth parameters. This indicates that the search for the hash symbol must begin at position 3 in the source string. Then returns the number 10, which is the position of the fourth occurrence of the hash symbol when the search begins at position 3.

The SUBSTR Function (Substring)

The SUBSTR function extracts and returns a segment from a given source string. It extracts a substring of a specified length from the source string beginning at a given position. If the start position is larger than the length of the source string, null is returned. If the number of characters to extract from a given start position is greater than the length of the source string, the segment returned is the substring from the start position to the end of the string.
Numeric and date literals are automatically cast as characters when they occur as parameters to the SUBSTR function. Numeric and date expressions are evaluated before being converted to strings ready to be searched.

The SUBSTR function takes three parameters, with the first two being mandatory.
Its syntax is SUBSTR(*source string*, *start position*, [*number of characters to extract*]). The default number of characters to extract is equal to the number of characters from the *start position* to the end of the *source string*.

```
Query 1: select substr(10000-3,3,2) from dual
Query 2: select substr(sysdate,4,3) from dual
Query 3: select substr('1#3#5#7#9#',5) from dual
Query 4: select substr('1#3#5#7#9#',5,6) from dual
Query 5: select substr('1#3#5#7#9#',-3,2) from dual
```

Query 1 returns substring 97.
Query 2 Assume that the current system date is 17-DEC-07. The search for the substring begins at position 4 and the three characters from that position onward are extracted, yielding the substring DEC.
Query 3 returns 5#7#9#.
Query 4 returns 5#7#9#.
Query 5 has the number –3 as its start position. The negative start position parameter instructs Oracle to commence searching 3 characters from the end of the string. Therefore start position is three characters from the end of the string, which is position 8. The third parameter is 2, which results in the substring #9 being returned.

The REPLACE Function

The REPLACE function replaces all occurrences of a search item in a source string with a replacement term and returns the modified source string. If the length of the replacement term is different from that of the search item, then the lengths of the returned and source strings will be different. If the search string is not found, the source string is returned unchanged. Numeric and date literals and expressions are evaluated before being implicitly cast as characters when they occur as parameters to the REPLACE function.
The REPLACE function takes three parameters, with the first two being mandatory.
Its syntax is REPLACE(*source string*, *search item*, [*replacement term*]).
If the *replacement term* parameter is omitted, each occurrence of the *search item* is removed from the *source string*. In other words, the *search item* is replaced by an empty string.

```
Query 1: select replace(10000-3,'9','85') from dual
Query 2: select replace(sysdate, 'DEC','NOV') from dual
Query 3: select replace('1#3#5#7#9#','#','->') from dual
Query 4: select replace('1#3#5#7#9#','#') from dual
```

Query 1 returns “8585857.”
Query 2 Assume that the current system date is 17-DEC-07. The search string “DEC” occurs once in the source string and is replaced with the characters “NOV,” yielding the result 17-NOV-07.
Query 3 returns 1->3->5->7->9->.
Query 4 does not specify a replacement string. The default behavior is therefore to replace the search string with an empty string which, in effect, removes the search character completely from the source, resulting in the string “13579” being returned.

Using Numeric Functions

A significant differentiator between numeric and other functions is that they accept and return only numeric data. Oracle provides three important *numeric single-row functions*: ROUND, TRUNC, and MOD.

The Numeric ROUND Function

The ROUND function performs a rounding operation on a numeric value based on the decimal precision specified. The value returned is either rounded up or down based on the numeric value of the significant digit at the specified decimal precision position. If the specified decimal precision is *n*, the digit significant to the rounding is found (*n* + 1) places to the RIGHT of the decimal point. If it is negative, the digit significant to the rounding is found *n* places to the LEFT of the decimal point. If the numeric value of the significant digit is greater than or equal to 5, a “round up” occurs, else a “round down” occurs.
The ROUND function takes two parameters. Its syntax is ROUND(*source number,decimal precision*). The *source number* parameter represents any numeric literal, column, or expression. The *decimal precision* parameter specifies the degree of rounding and is optional. If the *decimal precision* parameter is absent, the default degree of rounding is zero, which means the source is rounded to the nearest whole number.

```
Query 1: select round(1601.916718,1) from dual
Query 2: select round(1601.916718,2) from dual
Query 3: select round(1601.916718,-3) from dual
Query 4: select round(1601.916718) from dual
```

Query 1 returns 1601.9.
Query 2 returns 1601.92.
Query 3 The decimal precision parameter of the is –3. Since it is negative, the digit significant for rounding is found 3 places to the left of the decimal point, at the hundreds digit, which is 6. Since the hundreds unit is 6, rounding up occurs and the number returned is 2000.
Query 4 has dispensed with the decimal precision parameter. This implies that rounding is done to the nearest whole number. Since the tenth unit is 9, the number is rounded up and 1602 is returned.

The Numeric TRUNC Function (Truncate)

The TRUNC function performs a truncation operation on a numeric value based on the decimal precision specified. A numeric truncation is different from rounding because the resulting value drops the numbers at the decimal precision specified and does not attempt to round up or down if the decimal precision is positive. However, if the decimal precision specified (*n*) is negative, the input value is zeroed down from the *n*th decimal position.

The TRUNC function takes two parameters. Its syntax is TRUNC (*source number, decimal precision*). *Source number* represents any numeric literal, column, or expression. *Decimal precision* specifies the degree of truncation and is optional. If the *decimal precision* parameter is absent, the default degree of rounding is zero, which means the *source number* is truncated to the nearest whole number. If the *decimal precision* parameter is 1, then the *source number* is truncated at its tenths unit. If it is 2, it is truncated at its hundredths unit, and so on.

```
Query 1: select trunc(1601.916718,1) from dual
Query 2: select trunc(1601.916718,2) from dual
Query 3: select trunc(1601.916718,-3) from dual
Query 4: select trunc(1601.916718) from dual
```

Query 1 returns 1601.9.
Query 2 The *decimal precision* parameter (*n*) is 2, so it returns 1601.91.
Query 3 specifies a negative number (−3) as its decimal precision. Three places to the left of the decimal point implies that the truncation happens at the hundreds digit. Therefore, the *source number* is zeroed down from its hundreds digit (6) and the number returned is 1000.
Query 4 does not have a *decimal precision* parameter implying that truncation is done at the whole number degree of precision. The number returned is 1601.

The MOD Function (Modulus)

The MOD function returns the numeric remainder of a division operation. If the divisor is zero, no division by zero error is returned and the MOD function returns a zero instead. If the divisor is larger than the dividend, then the MOD function returns the dividend as its result. This is because it divides zero times into the divisor, leaving the remainder equal to the dividend.
The MOD function takes two parameters. Its syntax is MOD(*dividend, divisor*).
The *dividend* and *divisor* parameters represent a numeric literal, column, or expression, which may be negative or positive.

```
Query 1: select mod(6,2) from dual
Query 2: select mod(5,3) from dual
Query 3: select mod(7,35) from dual
Query 4: select mod(5.2,3) from dual
```

Query 1 returns 0.
Query 2 returns 2.
Query 3 attempts to divide 7 by 35. Since the *divisor* is larger than the *dividend*, the number 7 is returned.
Query 4 has an improper fraction as the *dividend*. Dividing 5.2 by 3 yields 1 with remainder 2.2.

Working with Dates

Date arithmetic and the *date manipulation functions*: ADD_MONTHS, MONTHS_BETWEEN, LAST_DAY, NEXT_DAY, ROUND, and TRUNC.

Date Storage in the Database

Dates are stored internally in a numeric format that supports the storage of century, year, month, and day details, as well as time information such as hours, minutes, and seconds. These date attributes are available for every literal, column value, or expression that is of date data type.
When date information is accessed from a table, the default format of the results comprises two digits that represent the day, a three-letter abbreviation of the month, and two digits representing the year component. By default, these components are separated with hyphens in SQL*Plus and forward slashes in SQL Developer.

Although the century component is not displayed by default, it is stored in the database when the date value is inserted or updated and is available for retrieval.
The format in which a date is displayed is referred to as its format mask.

Date Format Masks

DD Day of the month
MON Month of the year
YY Two-digit year
YYYY Four-digit year including century
RR Two-digit year (Year 2000–compliant)
CC Two-digit century
HH Hours with AM and PM
HH24 Twenty-four-hour time
MI Minutes
SS Seconds

The DD-MON-RR format mask is the default for display and input. When inserting or updating date information, the century component is obtained from the SYSDATE function if it is not supplied. The RR date format mask differs from the YY format mask since it is may be used to specify different centuries based on the current and specified years. The century component assigned to a date with its year specified with the RR date format may be better understood by considering the following principles:

- If the two digits of the current year and specified year lie between 0 and 49, the current century is returned. Suppose the present date is 02-JUN-2007. The century returned for the date 24-JUL-04 in DD-MON-RR format is 20.
- If the two digits of the current year lie between 0 and 49 and the specified year falls between 50 and 99, the previous century is returned. Suppose the current date is 02-JUN-2007. The century returned for 24-JUL-94 is 19.
- If the two digits of the current and specified years lie between 50 and 99, the current century is returned by default. Suppose the current date is 02-JUN-1975. The century returned for 24-JUL-94 is 19.
- If the two digits of the current year lie between 50 and 99 and the specified year falls between 0 and 49, the next century is returned. Suppose the current date is 02-JUN-1975. The century returned for 24-JUL-07 is 20.

The SYSDATE Function

The SYSDATE function takes no parameters and returns the current system date and time according to the database server. By default the SYSDATE function returns the DD-MON-RR components of the current system date. It is important to remember that SYSDATE does not return the date and time as specified by your local system clock. If the database server is located in a different time zone from a client querying the database, the date and time returned will differ from the local operating system clock on the client machine. The query to retrieve the database server date is as follows:

```
select sysdate from dual
```

Date Arithmetic

Date1 – Date2 = Num1
Date1 – Num1 = Date2
Date1 = Date2 + Num1

A date can be subtracted from another date. The difference between two date items represents the number of days between them. Any number, including fractions, may be added to or subtracted from a date item. In this context the number represents a number of days. The sum or difference between a number and a date item always returns a date item. This principle implies that adding, multiplying, or dividing two date items is not permitted.

Using Date Functions

The *date manipulation* functions provide a reliable and accurate means of working with date items. These functions provide such ease and flexibility for date manipulation that many integration specialists, database administrators, and other developers make frequent use of them.

The MONTHS_BETWEEN Function

The MONTHS_BETWEEN function returns a numeric value representing the number of months between two date values. Date literals in the format DD-MONRR or DD-MON-YYYY are automatically cast as date items when they occur as parameters to the MONTHS_BETWEEN function.

The MONTHS_BETWEEN function takes two mandatory parameters.

Its syntax is MONTHS_BETWEEN(*start date, end date*).

The function computes the difference in months between *start date* and *end date*. If the *end date* occurs before the *start date*, a negative number is returned. The difference between the two date parameters may consist of a whole number and a fractional component. The whole number represents the number of months between the two dates. The fractional component represents the days and time remaining after the integer difference between years and months is calculated and is based on a 31-day month. A whole number with no fractional part is returned if the day components of the dates being compared are either the same or the last day of their respective months.

```
Query 1: select sysdate, sysdate+31, sysdate+62, sysdate+91,
months_between(sysdate+91, sysdate) from dual
Query 2: select months_between('29-mar-2008','28-feb-2008') from dual
Query 3: select months_between('29-mar-2008','28-feb-2008') * 31 from dual
Query 4: select months_between(to_date('29-feb-2008'),
to_date('28-feb-2008 12:00:00','dd-mon-yyyy hh24:mi:ss'))* 31 from dual;
```

Assume that the current date is 29-DEC-2007.

The first expression in query 1 returns the number 1, as the month between 29-DEC-2007 is 29-JAN-2008 (31 days later). The second expression similarly returns 2 months between 29-DEC-2007 and 29-FEB-2008 (62 days later). Since February 2008 has 29 days, 91 days must be added to 29-DEC-2007 to get 29-MAR-2008, and the MONTHS_BETWEEN (29-MAR-2008, 29-DEC-2007) function returns exactly three months in the third expression in query 1.

Query 2 implicitly converts the date literals into date items of the format DD-MON-YYYY. Since no time information is provided, Oracle assumes the time to be midnight on both days, or 00:00:00. The MONTHS_BETWEEN function returns approximately 1.03225806. The whole number component indicates that there is one month between these two dates. Closer examination of the fractional component interestingly reveals that there is exactly one month between 28-MAR-2008 and 28-FEB-2008. The fractional component must therefore represent the one-day difference. It would include differences in hours, minutes, and seconds as well, but for this example, the time components are identical. Multiplying 0.03225806 by 31 returns 1, since the fractional component returned by MONTHS_BETWEEN is based on a 31-day month.

Query 3 returns the whole number 32.

Query 4 demonstrates how the time component is factored into the computation by the MONTHS_BETWEEN function, which returns approximately 0.016129. There is a 12-hour difference between the start and end date parameters, so zero months between them is correct. Multiplying the fractional part by 31 yields 0.5 days, which corresponds to the 12-hour difference.

The ADD_MONTHS Function

The ADD_MONTHS function returns a date item calculated by adding a specified number of months to a given date value. Date literals in the format DD-MON-RR or DD-MON-YYYY are automatically cast as date items when they occur as parameters to the ADD_MONTHS function.

The ADD_MONTHS function takes two mandatory parameters.

Its syntax is ADD_MONTHS (*start date, number of months*).

The function computes the target date after adding the specified number of months to the *start date*. The number of months may be negative, resulting in a target date earlier than the start date being returned. The *number of months* may be fractional, but the fractional component is ignored and the integer component is used.

Since the number of months added in the third query is –12, the date 07-APR-2008 is returned, which is 12 months prior to the start date.

The NEXT_DAY Function

The NEXT_DAY function returns the date when the next occurrence of a specified day of the week occurs. Literals that may be implicitly cast as date items are acceptable when they occur as parameters to the NEXT_DAY function.

The NEXT_DAY function takes two mandatory parameters.

Its syntax is NEXT_DAY (*start date, day of the week*).

The function computes the date on which the *day of the week* parameter next occurs after the *start date*. The *day of the week* parameter may be either a character value or an integer value. The acceptable values are determined by the NLS_DATE_LANGUAGE database parameter but the default values are at least the first three characters of the day name or integer values, where 1 represents Sunday, 2 represents Monday, and so on. The character values representing the days of the week may be specified in any case. The short name may be longer than three characters, for example, Sunday may be referenced as sun, sund, sunda or Sunday.

The LAST_DAY Function

The LAST_DAY function returns the date of the last day in the month a specified day belongs to. Literals that may be implicitly cast as date items are acceptable when they occur as parameters to the LAST_DAY function.

The LAST_DAY function takes one mandatory parameter.

Its syntax is LAST_DAY(*start date*). The function extracts the month that the *start date* parameter belongs to and calculates the date of the last day of that month.

The Date ROUND Function

The date ROUND function performs a rounding operation on a value based on a specified date precision format. The value returned is either rounded up or down to the nearest date precision format.

The date ROUND function takes one mandatory and one optional parameter.
Its syntax is ROUND(*source date, [date precision format*]).
The *source date* parameter represents any value that can be implicitly converted into a date item. The *date precision format* parameter specifies the degree of rounding and is optional. If it is absent, the default degree of rounding is *day*. This means the *source date* is rounded to the nearest day. The *date precision formats* include *century* (CC), *year* (YYYY), *quarter* (Q), *month* (MM), *week* (W), *day* (DD), *hour* (HH), and *minute* (MI). Rounding up to *century* is equivalent to adding one to the current century. Rounding up to the next month occurs if the *day* component is greater than 16, else rounding down to the beginning of the current month occurs. If the month falls between one and six, then rounding to *year* returns the date at the beginning of the current year, else it returns the date at the beginning of the following year.

The Date TRUNC Function

The date TRUNC function performs a truncation operation on a date value based on a specified date precision format.
The date TRUNC function takes one mandatory and one optional parameter.
Its syntax is TRUNC(*source date, [date precision format*]).
The *source date* parameter represents any value that can be implicitly converted into a date item. The *date precision format* parameter specifies the degree of truncation and is optional. If it is absent, the default degree of truncation is *day*. This means that any time component of the *source date* is set to midnight or 00:00:00 (00 hours, 00 minutes and 00 seconds). Truncating at the month level sets the date of the *source date* to the first day of the month. Truncating at the year level returns the date at the beginning of the current year.

TWO-MINUTE DRILL

Describe Various Types of Functions Available in SQL

❑ Functions accept zero or more input parameters but always return one result of a predetermined data type.

❑ Single-row functions execute once for each row selected, while multiple-row functions execute once for the entire set of rows queried.

❑ Character functions are either case-conversion or character-manipulation functions.

Use Character, Number, and Date Functions in SELECT Statements

❑ The INITCAP function accepts a string of characters and returns each word in title case.

❑ The function that computes the number of characters in a string including spaces and special characters is the LENGTH function.

❑ The INSTR function returns the positional location of the *nth* occurrence of a specified string of characters in a source string.

❑ The SUBSTR function extracts and returns a segment from a given source string.

❑ The REPLACE function substitutes each occurrence of a search item in the source string with a replacement term and returns the modified source string.

❑ A modulus operation returns the remainder of a division operation and is available via the MOD function.

❑ The numeric ROUND function rounds numbers either up or down to the specified degree of precision.

❑ The SYSDATE function is traditionally executed against the DUAL table and returns current date and time of the database server.

❑ Date types store century, year, month, day, hour, minutes, and seconds information.

❑ The difference between two date items is always a number that represents the number of days between these two items.

❑ Any number, including fractions, may be added to or subtracted from a date item and in this context the number represents a specified number of days.

❑ The MONTHS_BETWEEN function computes the number of months between two given date parameters and is based on a 31-day month.

❑ The LAST_DAY function is used to obtain the last day in a month given any valid date item.

SELF TEST

Describe Various Types of Functions Available in SQL

1. Which statements regarding single-row functions are true? (Choose all that apply.)
A. They may return more than one result.
B. They execute once for each record processed.
C. They may have zero or more input parameters.
D. They must have at least one mandatory parameter.

2. Which of these are single-row character-case conversion functions? (Choose all that apply.)
A. LOWER
B. SMALLER
C. INITCASE
D. INITCAP

Use Character, Number, and Date Functions in SELECT Statements

3. What value is returned after executing the following statement:
SELECT LENGTH('How_long_is_a_piece_of_string?') FROM DUAL; (Choose the best answer.)
A. 29
B. 30
C. 24
D. None of the above

4. What value is returned after executing the following statement:
SELECT SUBSTR('How_long_is_a_piece_of_string?', 5,4) FROM DUAL; (Choose the best answer.)
A. long
B. _long
C. string?
D. None of the above

5. What value is returned after executing the following statement?
SELECT INSTR('How_long_is_a_piece_of_string?','_',5,3) FROM DUAL; (Choose the best answer.)
A. 4
B. 14
C. 12
D. None of the above

6. What value is returned after executing the following statement?

SELECT REPLACE('How_long_is_a_piece_of_string?','_','') FROM DUAL; (Choose the best answer.)
A. How long is a piece of string?
B. How_long_is_a_piece_of_string?
C. Howlongisapieceofstring?
D. None of the above

7. What value is returned after executing the following statement?
SELECT MOD(14,3) FROM DUAL; (Choose the best answer.)
A. 3
B. 42
C. 2
D. None of the above

8. Assuming SYSDATE=07-JUN-1996 12:05pm, what value is returned after executing the following statement?
SELECT ADD_MONTHS(SYSDATE,-1) FROM DUAL; (Choose the best answer.)
A. 07-MAY-1996 12:05pm
B. 06-JUN-1996 12:05pm
C. 07-JUL-1996 12:05pm
D. None of the above

9. What value is returned after executing the following statement? Take note that 01-JAN-2009 occurs on a Thursday. (Choose the best answer.)
SELECT NEXT_DAY('01-JAN-2009','wed') FROM DUAL;
A. 07-JAN-2009
B. 31-JAN-2009
C. Wednesday
D. None of the above

10. Assuming SYSDATE=30-DEC-2007, what value is returned after executing the following statement?
SELECT TRUNC(SYSDATE,'YEAR') FROM DUAL; (Choose the best answer.)
A. 31-DEC-2007
B. 01-JAN-2008
C. 01-JAN-2007
D. None of the above

LAB QUESTION

Several quotations were requested for prices on color printers. The supplier information is not available from the usual source, but you know that the supplier identification number is embedded in the CATALOG_URL column from the PRODUCT_INFORMATION table. You are required to retrieve the PRODUCT_NAME and CATALOG_URL values and to extract the supplier number from the CATALOG_URL column for all products which have both the words COLOR and PRINTER in the PRODUCT_DESCRIPTION column stored in any case.

SELF TEST ANSWERS

Describe Various Types of Functions Available in SQL

1. ✹✓ **B** and **C.** Single-row functions execute once for every record selected in a dataset and may either take no input parameters, like SYSDATE, or many input parameters.
✹➢ **A** and **D** are incorrect because a function by definition returns only one result and there are many functions with no parameters.

2. ✹✓ **A** and **D.** The LOWER function converts the case of the input string parameter to its lowercase equivalent, while INITCAP converts the given input parameter to title case.
✹➢ **B** and **C** are not valid function names.

Use Character, Number, and Date Functions in SELECT Statements

3. ✹✓ **B.** The LENGTH function computes the number of characters in a given input string including spaces, tabs, punctuation mark, and other nonprintable special characters.
✹➢ **A, C,** and **D** are incorrect.

4. ✹✓ **A.** The SUBSTR function extracts a four-character substring from the given input string starting with and including the fifth character. The characters at positions 1 to 4 are How_. Starting with the character at position 5, the next four characters form the word "long."
✹➢ **B, C,** and **D** are incorrect because **B** is a five-character substring beginning at position 4, while ring?, which is also five characters long, starts five characters from the end of the given string.

5. ✹✓ **B.** The INSTR function returns the position that the *nth* occurrence of the search string may be found after starting the search from a given start position. The search string is the underscore character, and the third occurrence of this character starting from position 5 in the source string occurs at position 14.
✹➢ **A, C,** and **D** are incorrect since position 4 is the first occurrence of the search string and position 12 is the third occurrence if the search began at position 1.

6. ✹✓ **C.** All occurrences of the underscore character are replaced by an empty string, which removes them from the string.
✹➢ **A, B,** and **D** are incorrect. **A** is incorrect because the underscore characters are not replaced by spaces, and **B** does not change the source string.

7. ✹✓ **C.** When 14 is divided by 3, the answer is 4 with remainder 2.
✹➢ **A, B,** and **D** are incorrect.

8. ✹✓ **A.** The minus one parameter indicates to the ADD_MONTHS function that the date to be returned must be one month prior to the given date.
✹➢ **B, C,** and **D** are incorrect. **B** is one day and not one month prior to the given date. **C** is one month after the given date.

9. ✹✓ **A.** Since the first of January 2009 falls on a Thursday, the date of the following Wednesday is six days later.
✹➢ **B, C,** and **D** are incorrect. **B** returns the last day of the month in which the given date falls, and **C** returns a character string instead of a date.

10. ✹✓ **C.** The date TRUNC function does not perform rounding and since the degree of truncation is YEAR, the day and month components of the given date are ignored and the first day of the year it belongs to is returned.
✹➢ **A, B,** and **D** are incorrect. **A** returns the last day in the month in which the given date occurs, and **B** returns a result achieved by rounding instead of truncation.

LAB ANSWER

```
SELECT PRODUCT_NAME, CATALOG_URL, SUBSTR(CATALOG_URL, 17, 6) SUPPLIER
FROM PRODUCT_INFORMATION
WHERE UPPER(PRODUCT_DESCRIPTION) LIKE '%COLOR%' AND
UPPER(PRODUCT_DESCRIPTION) LIKE '%PRINTER%'
```

Chapter 4: Using Conversion Functions and Conditional Expressions

Sometimes data is not available in the exact format a function is defined to accept, resulting in a data type mismatch. To avoid mismatch errors, Oracle implicitly converts compatible data types. Explicit conversion functions, which are used for reliable data type conversions.

The concept of nesting functions is defined and a category of general functions aimed at simplifying interactions with NULL values is introduced. These include the NVL, NVL2, NULLIF, and COALESCE functions.

Conditional logic, or the ability to display different results depending on data values, is exposed by the conditional functions CASE and DECODE. These functions provide *if-then-else* logic in the context of a SQL query.

Types of Conversion Functions Available in SQL

SQL conversion *functions* are single row functions designed to alter the nature of the data type of a column value, expression or literal. *TO_CHAR, TO_NUMBER* and *TO_DATE* are the three most widely used conversion functions. The TO_CHAR function converts numeric and date information into characters, while TO_NUMBER and TO_DATE convert character data into numbers and dates, respectively.

Conversion Functions

Oracle allows columns to be defined with data types. Table definitions are obtained using the DESCRIBE command. Each column has an associated data type that constrains the nature of the data it can store. A NUMBER column cannot store character information. A DATE column cannot store random characters or numbers. However, the character equivalents of both number and date information can be stored in a VARCHAR2 field. If a function that accepts a character input parameter finds a number instead, Oracle automatically converts it into its character equivalent. If a function that accepts a number or a date parameter encounters a character value, there are specific conditions under which automatic data type conversion occurs. DATE and NUMBER data types are very strict compared to VARCHAR2 and CHAR. Although implicit data type conversions are available, it is more reliable to explicitly convert values from one data type to another using single-row conversion functions.

When numeric values are supplied as input to functions expecting character parameters, implicit data type conversion ensures that they are treated as character values. Similarly, character strings consisting of numeric digits are implicitly converted into numeric values if possible when a data type mismatch occurs. But be wary of implicit conversions. There are some cases when it does not work as expected, as in the following WHERE clause. Consider limiting data from a table T based on a character column C, which contains the string '100'. The condition clause WHERE C='100' works as you might expect, but the condition WHERE C=100 returns an invalid number error.

Implicit Data Type Conversion

Values that do not share identical data types with function parameters are *implicitly converted* to the required format if possible. VARCHAR2 and CHAR data types are collectively referred to as character types. Character fields are flexible and allow the storage of almost any type of information. Therefore, DATE and NUMBER values can easily be converted to their character equivalents. These conversions are known as *number to character* and *date to character* conversions.

```
Query 1: select length(1234567890) from dual
Query 2: select length(SYSDATE) from dual
```

Both queries use the LENGTH function, which takes a character string parameter. The number 1234567890 in query 1 is implicitly converted into a character string, '1234567890', before being evaluated by the LENGTH function, which returns the number 10. Query 2 first evaluates the SYSDATE function, which is assumed to be 07-APR-38. This date is implicitly converted into the character string '07-APR-38', and the LENGTH function returns the number 9.

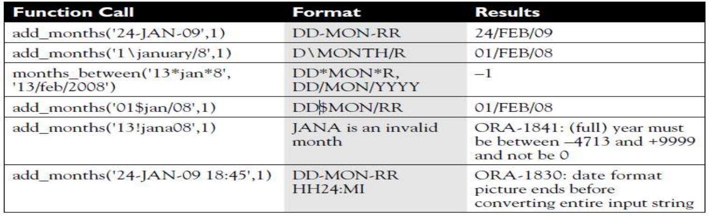
It is uncommon for character data to be implicitly converted into numeric data types since the only condition under which this occurs is if the character data represents a valid number. The character string '11' will be implicitly converted to a number, but '11.123.456' will not be.

```
Query 3: select mod('11',2) from dual
Query 4: select mod('11.123',2) from dual
Query 5: select mod('11.123.456',2) from dual
Query 6: select mod('$11',2) from dual
```

Queries 3 and 4 implicitly convert the character strings '11' and '11.123' into the numbers 11 and 11.123, respectively, before the MOD function evaluates them and returns the results 1 and 1.123. Query 5 returns the error “ORA-1722: invalid number,” when Oracle tries to perform an implicit *character to number* conversion. It fails because the string '11.123.456' is not a valid number. Query 6 also fails with the invalid number error, since the dollar symbol cannot be implicitly converted into a number.

Implicit *character to date* conversions are possible when the character string conforms to the following date patterns: [D|DD] *separator1* [MON|MONTH] *separator2* [R|RR|YY|YYYY]. D and DD represent a single and 2-digit day of the month. MON is a 3-character abbreviation, while MONTH is the full name for a month. R and RR represent a single and 2-digit year. YY and YYYY represent a 2- and 4-digit year, respectively. The *separator1* and *separator2* elements may be most punctuation marks, spaces, and tabs.

Examples of Implicit Character to Date Conversion



Function Call	Format	Results
add_months('24-JAN-09',1)	DD-MON-RR	24/FEB/09
add_months('1\january/8',1)	D\MONTH/R	01/FEB/08
months_between('13*jan*8', '13/feb/2008')	DD*MON*R, DD/MON/YYYY	–1
add_months('01$jan/08',1)	DD$MON/RR	01/FEB/08
add_months('13!jana08',1)	JANA is an invalid month	ORA-1841: (full) year must be between –4713 and +9999 and not be 0
add_months('24-JAN-09 18:45',1)	DD-MON-RR HH24:MI	ORA-1830: date format picture ends before converting entire input string

Explicit Data Type Conversion

Oracle offers many functions to convert items from one data type to another, known as *explicit* data type conversion functions. These return a value guaranteed to be the type required and offer a safe and reliable method of converting data items.

NUMBER and DATE items can be converted explicitly into character items using the TO_CHAR function. A character string can be explicitly changed into a NUMBER using the TO_NUMBER function. The TO_DATE function is used to convert character strings into DATE items. Oracle’s format masks enable a wide range of control over *character to number* and *character to date* conversions.

Syntax of Explicit Data Type Conversion Functions

TO_NUMBER(*char1,* [format mask], [nls_parameters]) = *num1*
TO_CHAR(*num1*, [format mask], [nls_parameters]) = *char1*
TO_DATE(*char1,* [format mask], [nls_parameters]) = *date1*
TO_CHAR(*date1*, [format mask], [nls_parameters]) = *char1*

TO_CHAR, TO_NUMBER, and TO_DATE Conversion Functions

TO_CHAR conversion is divided into two types of items to characters: DATE and NUMBER. This separation is warranted by the availability of different format masks for controlling conversion to character values. These conversion functions exist alongside many others but tend to be the most widely used.

Using the Conversion Functions

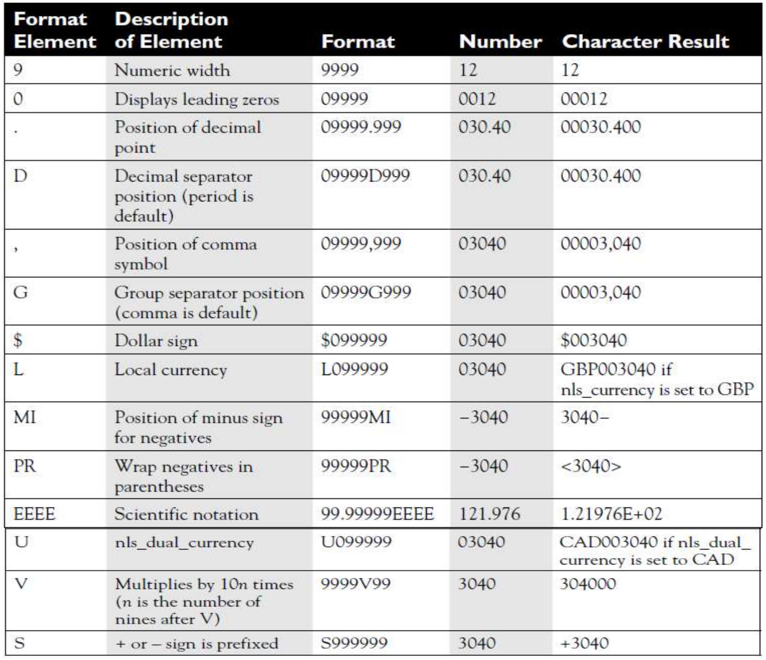
Many situations demand the use of conversion functions. They may range from formatting DATE fields in a report to ensuring that numeric digits extracted from character fields are correctly converted into numbers before applying them in an arithmetic expression.

Converting Numbers to Characters Using the TO_CHAR Function

The TO_CHAR function returns an item of data type VARCHAR2. When applied to items of type NUMBER, several formatting options are available. The syntax is as follows:

TO_CHAR(*number1, [format], [nls_parameter]),*

The *number1* parameter is mandatory and must be a value that either is or can be implicitly converted into a number. The optional *format* parameter may be used to specify numeric formatting information like width, currency symbol, the position of a decimal point, and group (or thousands) separators and must be enclosed in single quotation marks.



Format Element	Description of Element	Format	Number	Character Result
9	Numeric width	9999	12	12
0	Displays leading zeros	09999	0012	00012
.	Position of decimal point	09999.999	030.40	00030.400
D	Decimal separator position (period is default)	09999D999	030.40	00030.400
,	Position of comma symbol	09999,999	03040	00003,040
G	Group separator position (comma is default)	09999G999	03040	00003,040
$	Dollar sign	$099999	03040	$003040
L	Local currency	L099999	03040	GBP003040 if nls_currency is set to GBP
MI	Position of minus sign for negatives	99999MI	−3040	3040−
PR	Wrap negatives in parentheses	99999PR	−3040	<3040>
EEEE	Scientific notation	99.99999EEEE	121.976	1.21976E+02
U	nls_dual_currency	U099999	03040	CAD003040 if nls_dual_currency is set to CAD
V	Multiplies by 10n times (*n* is the number of nines after V)	9999V99	3040	304000
S	+ or – sign is prefixed	S999999	3040	+3040

```
Query 1: select to_char(00001)||' is a special number' from dual;
Query 2: select to_char(00001,'0999999')||' is a special number'from dual;
```

Query 1 evaluates the number 00001, removes the leading zeros, converts the number 1 into the character '1' and returns the character string ‘1 is a special number’.

Query 2 applies the numeric format mask '0999999' to the number 00001, converting it into the character string '0000001'. After concatenation to the character literals, the string returned is ‘0000001 is a special number’. The zero and 6 nines in the format mask indicate to the TO_CHAR function that leading zeros must be displayed and that the display width must be set to seven characters. Therefore, the string returned by the TO_CHAR function contains seven characters.

Converting numbers into characters is a reliable way to ensure that functions and general SQL syntax, which expects character input, do not return errors when numbers are encountered. Converting numbers into character strings is common when numeric data must be formatted for reporting purposes. The format masks that support currency, thousands separators, and decimal point separators are frequently used when presenting financial data.

Converting Dates to Characters Using the TO_CHAR Function

You can take advantage of a variety of format models to convert DATE items into almost any character representation of a date using TO_CHAR. Its syntax is as follows:

TO_CHAR(*date1, [format], [nls_parameter]),*

Only the *date1* parameter is mandatory and must take the form of a value that can be implicitly converted to a date. The optional *format* parameter is case sensitive and must be enclosed in single quotes. The format mask specifies which date elements are extracted and whether the element should be described by a long or an abbreviated name. The names of days and months are automatically padded with spaces. These may be removed using a modifier to the format mask called the fill mode (*fm*) operator. By prefixing the format model with the letters fm, Oracle is instructed to trim all spaces from the names of days and months. There are many formatting options for dates being converted into characters.

Consider the following three queries:

```
Query 1: select to_char(sysdate)||' is today''s date' from dual;
Query 2: select to_char(sysdate,'Month')||'is a special time'from dual;
Query 3: select to_char(sysdate,'fmMonth')||'is a special time'from dual;
```

If the current system date is 03/JAN/09 and the default display format is DD/MON/RR, then query 1 returns the character string ‘03/JAN/09 is today's date’. There are two notable components in query 2.

First, only the month component of the current system date is extracted for conversion to a character type. Second, since the format mask is case sensitive and 'Month' appears in title case, the string returned is ‘January is a special time’. There is no need to add a space in front of the literal 'is a special time' since the TO_CHAR function automatically pads the name of the month with a space. If the format mask in query 2 was 'MONTH', the string returned would be ‘JANUARY is a special time’. The *fm* modifier is applied to query 3, and the resultant string is ‘January is a special time’. Note there is no space between January and the literal 'is a special time'.

Converting Dates into Characters Using the TO_CHAR Function

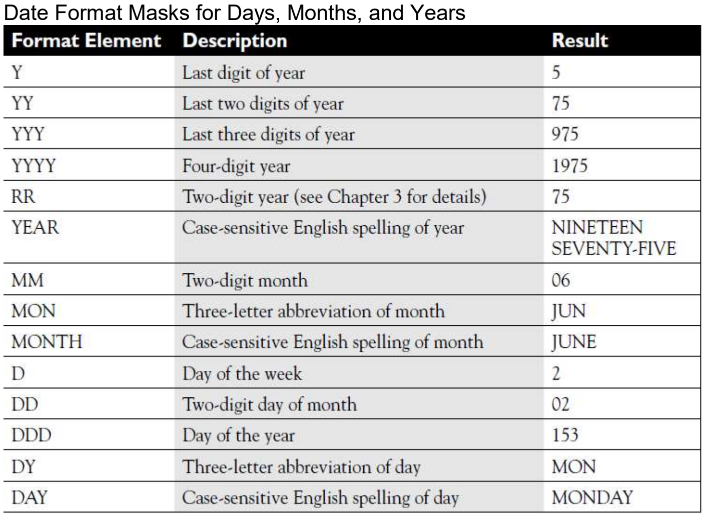
You are required to retrieve a list of FIRST_NAME and LAST_NAME values and an expression based on the HIRE_DATE column for employees hired on a Saturday. The expression must be aliased as START_DATE and a HIRE_DATE value of 17-FEB-1996 must return the following string:
Saturday, the 17th of February, One Thousand Nine Hundred Ninety-Six.

```
SELECT FIRST_NAME, LAST_NAME, TO_CHAR(HIRE_DATE, 'fmDay, "the "ddth "of " Month, Yyyysp.') START_DATE
FROM EMPLOYEES
WHERE TO_CHAR(HIRE_DATE,'fmDay') = 'Saturday'
```

Converting Characters to Dates Using the TO_DATE Function

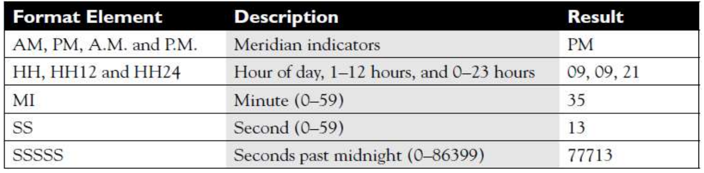
The TO_DATE function returns an item of type DATE. Character strings converted to dates may contain all or just a subset of the date time elements comprising a DATE. When strings with only a subset of the date time elements are converted, Oracle provides default values to construct a complete date. Components of character strings are associated with different date time elements using a format model or mask. The syntax is as follows:
TO_DATE(*string1, [format], [nls_parameter]),*
Only the *string1* parameter is mandatory and if no format mask is supplied, *string1* must take the form of a value that can be implicitly converted into a date. The optional *format* parameter is almost always used and is specified in single quotation marks. The TO_DATE function has an *fx* modifier which is similar to *fm* used with the TO_CHAR function. *fx* specifies an exact match for *string1* and the format mask. When the *fx* modifier is specified, character items that do not exactly match the format mask yield an error.

Date Format Masks for Days, Months, and Years



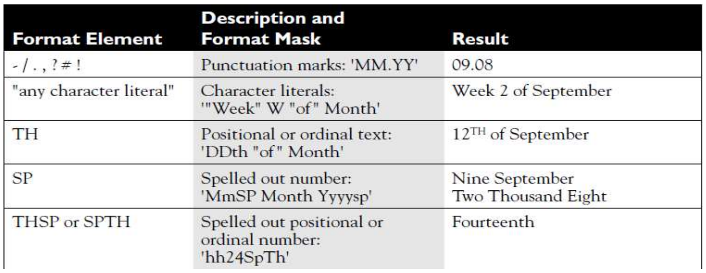
Format Element	Description	Result
Y	Last digit of year	5
YY	Last two digits of year	75
YYY	Last three digits of year	975
YYYY	Four-digit year	1975
RR	Two-digit year (see Chapter 3 for details)	75
YEAR	Case-sensitive English spelling of year	NINETEEN SEVENTY-FIVE
MM	Two-digit month	06
MON	Three-letter abbreviation of month	JUN
MONTH	Case-sensitive English spelling of month	JUNE
D	Day of the week	2
DD	Two-digit day of month	02
DDD	Day of the year	153
DY	Three-letter abbreviation of day	MON
DAY	Case-sensitive English spelling of day	MONDAY

Date Format Mask for Time Components



Format Element	Description	Result
AM, PM, A.M. and P.M.	Meridian indicators	PM
HH, HH12 and HH24	Hour of day, 1–12 hours, and 0–23 hours	09, 09, 21
MI	Minute (0–59)	35
SS	Second (0–59)	13
SSSSS	Seconds past midnight (0–86399)	77713

Miscellaneous Date Format Masks



Format Element	Description and Format Mask	Result
- / . , ? # !	Punctuation marks: 'MM.YY'	09.08
"any character literal"	Character literals: '"Week" W "of" Month'	Week 2 of September
TH	Positional or ordinal text: 'DDth "of" Month'	12^{TH} of September
SP	Spelled out number: 'MmSP Month Yyyysp'	Nine September Two Thousand Eight
THSP or SPTH	Spelled out positional or ordinal number: 'hh24SpTh'	Fourteenth

Consider the following five queries:

```
Query 1: select to_date('25-DEC-2010') from dual;
Query 2: select to_date('25-DEC') from dual;
Query 3: select to_date('25-DEC', 'DD-MON') from dual;
Query 4: select to_date('25-DEC-2010 18:03:45', 'DD-MON-YYYY HH24:MI:SS') from dual;
Query 5: select to_date('25-DEC-10', 'fxDD-MON-YYYY') from dual;
```

Query 1 evaluates the string 25-DEC-2010 and has sufficient information to implicitly convert it into a DATE item with a default mask of DD-MON-YYYY. The hyphen separator could be substituted with another punctuation character. Since no time components are provided, the time for this converted date is set to midnight or 00:00:00.
Query 2 cannot implicitly convert the string into a date because there is insufficient information and an “ORA-01840: input value is not long enough for date format” error is returned.
By supplying a format mask DD-MON to the string 25-DEC in query 3, Oracle can match the number 25 to DD and the abbreviated month name DEC to the MON component. Year and time components are absent, so the current year returned by the SYSDATE function is used and the time is set to midnight. If the current year is 2009, query 3 returns the date 25/DEC/09 00:00:00.
Query 4 performs a complete conversion of a string with all the date time elements present, and no default values are supplied by Oracle. Query 5 uses the *fx* modifier in its format mask. Since the year component of the string is 10 and the corresponding format mask is YYYY, the *fx* modifier results in an “ORA-01862: the numeric value does not match the length of the format item” error being returned.

Converting Characters to Numbers Using the TO_NUMBER Function

The TO_NUMBER function returns an item of type NUMBER. Character strings converted into numbers must be suitably formatted so that any nonnumeric components are translated or stripped away with an appropriate format mask.
The syntax is as follows:

TO_NUMBER(*string1, [format], [nls_parameter]),*

Only the *string1* parameter is mandatory and if no *format* mask is supplied, it must be a value that can be implicitly converted into a number. The optional *format* parameter is specified in single quotation marks.

```
Query 1: select to_number('$1,000.55') from dual;
Query 2: select to_number('$1,000.55','$999,999.99') from dual;
```

Query 1 cannot perform an implicit conversion to a number because of the dollar sign, comma, and period and returns the error, “ORA-1722: invalid number.”

Query 2 matches the dollar symbol, comma, and period from the string to the format mask and, although the numeric width is larger than the string width, the number 1000.55 is returned.

Apply Conditional Expressions in a SELECT Statement

These functions provide the language for dealing effectively with NULL values, and the *conditional functions*, which support conditional logic in expressions

Nested Functions

Nested functions use the output from one function as the input to another. Functions always return exactly one result. Therefore, you can reliably consider a function call in the same way as you would a literal value, when providing input parameters to a function. Single row functions can be nested to any level of depth. The general form of a function is as follows:

Function1(*parameter 1, parameter2,…*) = *result1*

Substituting function calls as parameters to other functions may lead to an expression such as the following:

F1(*param1.1*, F2(*param2.1*, *param2.2*, F3(*param3.1)*), *param1.3*)

Nested functions are first evaluated before their return values are used as parametric input to other functions. They are evaluated from the innermost to outermost levels.

The preceding expression is evaluated as follows:

1. F3(*param3.1*) is evaluated and its return value provides the third parameter to function F2 and may be called: *param2.3*.
2. F2(*param2.1,param2.2,param2.3*) is evaluated and its return value provides the second parameter to function F1 and is *param1.2.*
3. F1(*param1.1, param1.2, param1.3*) is evaluated and the result is returned to the calling program.

Function F3 is said to be nested three levels deep in this example. Consider the following query:

```
select length(to_char(to_date('28/10/09', 'DD/MM/RR'),'fmMonth'))from dual;
```

There are three functions in the SELECT list which, from inner to outer levels, are TO_DATE*,* TO_CHAR and *LENGTH.* The query is evaluated as follows:

1. The innermost function is evaluated first. TO_DATE('28/10/09','DD/MM/RR') converts the character string 28/10/09 into the DATE value 28-OCT-2009. The RR format mask is used for the year portion. Therefore, the century component
returned is the current century (the twenty-first), since the year component is between 0 and 49.
2. The second innermost function is evaluated next. TO_CHAR('28-OCT-2009', 'fmMonth') converts the given date based on the Month format mask and returns the character string October. The *fm* modifier trims trailing blank spaces from the name of the month.
3. Finally, the LENGTH('October') function is evaluated and the query returns the number 7.

General Functions

General functions simplify working with columns that potentially contain null values. These functions accept input parameters of all data types. The services they offer are primarily relevant to null values.

The functions examined in the next sections include the *NVL* function, which provides an alternative value to use if a null is encountered. The *NVL2* function performs a conditional evaluation of its first parameter and returns a value if a null is encountered and an alternative if the parameter is not null. The *NULLIF* function compares two terms and returns a null result if they are equal, otherwise it returns the first term. The *COALESCE* function accepts an unlimited number of parameters and returns the first nonnull parameter else it returns null.

The NVL Function

The NVL function evaluates whether a column or expression of any data type is null or not. If the term is null, an alternative not null value is returned; otherwise, the initial term is returned.

The NVL function takes two mandatory parameters. Its syntax is **NVL(*original, ifnull*)**, where *original* represents the term being tested and *ifnull* is the result returned if the *original* term evaluates to null. The data types of the *original* and *ifnull* parameters must always be compatible. They must either be of the same type, or it must be possible to implicitly convert *ifnull* to the type of the *original* parameter.

The NVL function returns a value with the same data type as the *original* parameter.

Consider the following three queries:

```
Query 1: select nvl(1234) from dual;
Query 2: select nvl(null,1234) from dual;
Query 3: select nvl(substr('abc',4),'No substring exists') from dual;
```

Since the NVL function takes two mandatory parameters, query 1 returns the error, “ORA-00909: invalid number of arguments.”

Query 2 returns 1234 after the null keyword is tested and found to be null.

Query 3 involves a nested SUBSTR function that attempts to extract the fourth character from a 3-character string. The
inner function returns null, leaving the NVL(null,'No substring exists') function to execute, which then returns the string ‘No substring exists’.

The NVL2 Function

The NVL2 function provides an enhancement to NVL but serves a very similar purpose. It evaluates whether a column or expression of any data type is null or not.

If the first term is not null, the second parameter is returned, else the third parameter is returned. Recall that the NVL function is different since it returns the original term if it is not null.

The NVL2 function takes three mandatory parameters. Its syntax is NVL2(*original, ifnotnull, ifnull*), where *original* represents the term being tested. *Ifnotnull* is returned if *original* is not null, and *ifnull* is returned if *original* is null. The data types of the *ifnotnull* and *ifnull* parameters must be compatible, and they cannot be of type LONG.

They must either be of the same type, or it must be possible to convert *ifnull* to the type of the *ifnotnull* parameter. The data type returned by the NVL2 function is the same as that of the *ifnotnull* parameter. Consider the following three queries:

```
Query 1: select nvl2(1234,1,'a string') from dual;
Query 2: select nvl2(null,1234,5678) from dual;
Query 3: select nvl2(substr('abc',2),'Not bc','No substring') from dual;
```

The *ifnotnull* term in query 1 is a number and the *ifnull* parameter is 'a string'. Since there is a data type incompatibility between them, an “ORA-01722: invalid number” error is returned.

Query 2 returns the *ifnull* parameter, which is 5678.

Query 3 extracts the characters “bc” using the SUBSTR function and the NVL2('bc',Not bc','No Substring') function is evaluated. The *ifnotnull* parameter, the string 'Not bc', is returned.

The NULLIF Function

The NULLIF function tests two terms for equality. If they are equal the function returns a null, else it returns the first of the two terms tested.

The NULLIF function takes two mandatory parameters of any data type. The syntax is NULLIF(*ifunequal, comparison_term*), where the parameters *ifunequal* and *comparison_term* are compared. If they are identical, then NULL is returned. If they differ, the *ifunequal* parameter is returned.

Consider the following three queries:

```
Query 1: select nullif(1234,1234) from dual;
Query 2: select nullif(1234,123+1) from dual;
Query 3: select nullif('24-JUL-2009','24-JUL-09') from dual;
```

Query 1 returns a null value since the parameters are identical.

The arithmetic equation in query 2 is not implicitly evaluated, and the NULLIF function finds 1234 different from 123+1, so it returns the *ifunequal* parameter, which is 1234.

The character literals in query 3 are not implicitly converted to DATE items and are compared as two character strings by the NULLIF function. Since the strings are of different lengths, the *ifunequal* parameter 24-JUL-2009 is returned.

Using NULLIF and NVL2 for Simple Conditional Logic

You are required to return a set of rows from the EMPLOYEES table with DEPARTMENT_ID values of 100. The set must also contain FIRST_NAME and LAST_NAME values and an expression aliased as NAME_LENGTHS. This expression must return the string 'Different Length' if the length of the FIRST_NAME differs from that of the LAST_NAME, else the string 'Same Length' must be returned.

SELECT FIRST_NAME, LAST_NAME, NVL2(NULLIF(LENGTH(LAST_NAME), LENGTH(FIRST_NAME)), 'Different Length', 'Same Length') NAME_LENGTHS
FROM EMPLOYEES
WHERE DEPARTMENT_ID=100

The COALESCE Function

The COALESCE function returns the first nonnull value from its parameter list. If all its parameters are null, then null is returned.

The COALESCE function takes two mandatory parameters and any number of optional parameters.

The syntax is COALESCE(*expr1, expr2,…,exprn*), where *expr1* is returned if it is not null, else *expr2* if it is not null, and so on. COALESCE is a general form of the NVL function.

COALESCE(expr1,expr2) = NVL(expr1,expr2)
COALESCE(expr1,expr2,expr3) = NVL(expr1,NVL(expr2,expr3))

The data type COALESCE returns if a not null value is found is the same as that of the first not null parameter. To avoid an “ORA-00932: inconsistent data types” error, all not null parameters must have data types compatible with the first not null parameter.

Consider the following three queries:

```
Query 1: select coalesce(null, null, null, 'a string') from dual;
Query 2: select coalesce(null, null, null) from dual;
Query 3: select coalesce(substr('abc',4),'Not bc','No substring') from dual;
```

Query 1 returns the fourth parameter: a string, since this is the first not null parameter encountered.

Query 2 returns null because all its parameters are null.

Query 3 evaluates its first parameter, which is a nested SUBSTR function, and finds it to be null. The second parameter is not null so the string 'Not bc' is returned.

Conditional Functions

Conditional logic, also known as *if-then-else* logic, refers to choosing a path of execution based on data values meeting certain conditions. *Conditional functions*, like DECODE and the CASE expression, return different values based on evaluating comparison conditions. These conditions are specified as parameters to the DECODE function and the CASE expression. The DECODE function is specific to Oracle, while the CASE expression is ANSI SQL compliant. An example of *if-then-else* logic is: if the country value is Brazil or Australia, then return Southern Hemisphere, else return Northern Hemisphere.

The DECODE Function

Although its name sounds mysterious, this function is straightforward. The DECODE function implements *if-then-else* conditional logic by testing its first two terms for equality and returns the third if they are equal and optionally returns another term if they are not.

The DECODE function takes at least three mandatory parameters, but can take many more.

The syntax of the function is DECODE(*expr1,comp1, iftrue1, [comp2,iftrue2...[compN,iftrueN]], [iffalse]*).

```
If expr1 = comp1 then return iftrue1
else if expr1 = comp2 then return iftrue2
...
...
else if expr1 = compN then return iftrueN
else return null | iffalse;
```

Expr1 is compared to *comp1.* If they are equal, then *iftrue1* is returned. If *expr1* is not equal to *comp1*, then what happens next depends on whether the optional parameters *comp2* and *iftrue2* are present. If they are, then *expr1* is compared to *comp2.* If they are equal, then *iftrue2* is returned. If not, what happens next depends on whether further *compn,iftruen* pairs exist, and the cycle continues until no comparison terms remain. If no matches have been found and if the *iffalse* parameter is defined, then *iffalse* is returned. If the *iffalse* parameter does not exist and no matches are found, a null value is returned.

All parameters to the DECODE function may be expressions. The return data type is the same as that of the first matching comparison item. The expression *expr1* is implicitly converted to the data type of the first comparison parameter *comp1*.

As the other comparison parameters *comp2…compn* are evaluated, they too are implicitly converted to the same data type as *comp1*. *Decode* considers two nulls to be equivalent, so if *expr1* is null and *comp3* is the first null comparison parameter encountered, then the corresponding result parameter *iftrue3* is returned.

```
Query 1: select decode(1234,123,'123 is a match') from dual;
Query 2: select decode(1234,123,'123 is a match','No match') from dual;
Query 3: select decode('search','comp1','true1', 'comp2','true2','search','true3',
substr('2search',2,6)),'true4', 'false') from dual;
```

Query 1 compares the number 1234 with the first comparison term 123. Since they are not equal, the first result term cannot be returned. Further, as there is no default *iffalse* parameter defined, a null is returned.

Query 2 is identical to the first except that an *iffalse* parameter is defined. Therefore, since 1234 is not equal to 123, the string ‘No match’ is returned.

Query 3 searches through the comparison parameters for a match. The character terms ‘comp1’ and ‘comp2’ are not equal to search, so the results true1 and true2 are not returned. A match is found in the third comparison term ‘comp3’ (parameter 6), which contains the string search. Therefore, the third result term *iftrue3* (parameter 7) containing the string ‘true3’ is returned.

Note that since a match has been found, no further searching takes place. So, although the fourth comparison term (parameter 8) is also a match to *expr1*, this expression is never evaluated, because a match was found in an earlier comparison term.

The CASE Expression

Virtually all third and fourth generation programming languages implement a *case* statement. Like the DECODE function, the CASE expression facilitates *if-then-else* conditional logic. There are two variants of the CASE expression. The *simple CASE expression* lists the conditional search item once, and equality to the search item is tested by each comparison expression. The *searched CASE expression* lists a separate condition for each comparison expression.

The CASE expression takes at least three mandatory parameters but can take many more. Its syntax depends on whether a simple or a searched CASE expression is used.

The syntax for the simple CASE expression is as follows:

CASE *search_expr*
WHEN *comparison_expr1* THEN *iftrue1*
[WHEN *comparison_expr2* THEN *iftrue2*
…
WHEN *comparison_exprN* THEN *iftrueN*
ELSE *iffalse*]
END

The simple CASE expression is enclosed within a CASE…END block and consists of at least one WHEN…THEN statement. In its simplest form, with one WHEN…THEN statement, the *search_expr* is compared with the *comparison_expr1*.

If they are equal, then the result *iftrue1* is returned. If not, a null value is returned unless an ELSE component is defined, in which case, the default *iffalse* value is returned.

When more than one WHEN…THEN statement exists in the CASE expression, searching for a matching comparison expression continues until a match is found.

The search, comparison, and result parameters can be column values, expressions, or literals but must all be of the same data type.

```
select
case substr(1234,1,3)
when '134' then '1234 is a match'
when '1235' then '1235 is a match'
when concat('1','23') then concat('1','23')||' is a match'
else 'no match'
end
from dual;
```

The search expression derived from the SUBSTR(1234,1,3) is the character string 123. The first WHEN...THEN statement compares the string 134 with 123. Since they are not equal, the result expression is not evaluated. The second WHEN…THEN statement compares the string 1235 with 123 and again, they are not equal. The third WHEN…THEN statement compares the results derived from the CONCAT('1','23') expression, which is 123, to the search expression. Since they are identical, the third results expression '123 is a match', is returned.

The syntax for the searched CASE expression is as follows:

CASE
WHEN *condition1* THEN *iftrue1*
[WHEN *condition2* THEN *iftrue2*
…
WHEN *conditionN* THEN *iftrueN*
ELSE *iffalse*]
END

The searched CASE expression is enclosed within a CASE…END block and consists of at least one WHEN…THEN statement. In its simplest form with one WHEN…THEN statement, *condition1* is evaluated; if it is true, then the result *iftrue1* is returned. If not, a null value is returned unless an ELSE component is defined, in which case the default *iffalse* value is returned. When more than one WHEN…THEN statement exists in the CASE expression, searching for a matching comparison expression continues until one is found.

The query to retrieve the identical set of results, using a searched CASE expression is listed next:

```
select last_name, hire_date,trunc(months_between(sysdate,hire_date)/12) years,
trunc(months_between(sysdate,hire_date)/60) "Years divided by 5",
case
when trunc(months_between(sysdate,hire_date)/60) < 1 then 'Intern'
when trunc(months_between(sysdate,hire_date)/60) < 2 then 'Junior'
when trunc(months_between(sysdate,hire_date)/60) < 3 then 'Intermediate'
when trunc(months_between(sysdate,hire_date)/60) < 4 then 'Senior'
else 'Furniture'
end Loyalty
from employees
where department_id in (60,10);
```

Using the DECODE Function

You are requested to query the LOCATIONS table for rows with the value US in the COUNTRY_ID column. An expression aliased as LOCATION_INFO is required to evaluate the STATE_PROVINCE column values and returns different information as per the following table. Sort the output based on the LOCATION_INFO expression.

Washington The string 'Headquarters'
Texas The string 'Oil Wells'
California The CITY column value
New Jersey The STREET_ADDRESS column value

```
SELECT DECODE(STATE_PROVINCE, 'Washington', 'Headquarters', 'Texas', 'Oil Wells', 'California', CITY, 'New Jersey',
STREET_ADDRESS) LOCATION_INFO
FROM EMPLOYEES
WHERE COUNTRY_ID='US'
ORDER BY LOCATION_INFO
```

TWO-MINUTE DRILL

Describe Various Types of Conversion Functions Available in SQL

❑ When values do not match the defined parameters of functions, Oracle attempts to convert them into the required data types. This is known as implicit conversion.

❑ Explicit conversion occurs when a function like TO_CHAR is invoked to change the data type of a value.

❑ The TO_CHAR function performs date to character and number to character data type conversions.

❑ Character items are explicitly transformed into date values using the TO_DATE conversion function.

❑ Character items are changed into number values using the TO_NUMBER conversion function.

Use the TO_CHAR, TO_NUMBER, and TO_DATE Conversion Functions

❑ The TO_CHAR function returns an item of type VARCHAR2.

❑ Format models or masks prescribe patterns that character strings must match to facilitate accurate and consistent conversion into number or date items.

❑ When the TO_CHAR function performs number to character conversions, the format mask can specify currency, numeric width, position of decimal operator, thousands separator, and many other formatting codes.

❑ The format masks available when TO_CHAR is used to convert character items to date include day, week, month, quarter, year, and century.

❑ Format masks must always be specified enclosed in single quotes.

❑ When performing date to character conversion, the format mask specifies which date elements are extracted and whether the element should be described by a long or abbreviated name.

❑ Character terms, like month and day names, extracted from dates with the TO_CHAR function are automatically padded with spaces that may be trimmed by prefixing the format mask with the *fm* modifier.

❑ The TO_DATE function has an *fx* modifier that specifies an exact match for the character string to be converted and the date format mask.

Apply Conditional Expressions in a SELECT Statement

❑ Nesting functions use the output from one function as the input to another.

❑ The NVL function either returns the original item unchanged or an alternative item if the initial term is null.

❑ The NVL2 function returns a new *if-null* item if the original item is null or an alternative *if-not-null* item if the original term is not null.

❑ The NULLIF function tests two terms for equality. If they are equal, the function returns null, else it returns the first of the two terms tested.

❑ The COALESCE function returns the first nonnull value from its parameter list. If all its parameters are null, then a null value is returned.

❑ The DECODE function implements *if-then-else* conditional logic by testing two terms for equality and returning the third term if they are equal or, optionally, some other term if they are not.

❑ There are two variants of the CASE expression used to facilitate *if-then-else* conditional logic: the simple CASE and searched CASE expressions.

SELF TEST

Choose all the correct answers for each question.

Describe Various Types of Conversion Functions Available in SQL

1. What type of conversion is performed by the following statement?
SELECT LENGTH(3.14285) FROM DUAL; (Choose the best answer.)
A. Explicit conversion
B. Implicit conversion
C. TO_NUMBER function conversion
D. None of the above

2. Choose any incorrect statements regarding conversion functions. (Choose all that apply.)
A. TO_CHAR may convert date items to character items.
B. TO_DATE may convert character items to date items.
C. TO_CHAR may convert numbers to character items.
D. TO_DATE may convert date items to character items.

Use the TO_CHAR, TO_NUMBER, and TO_DATE Conversion Functions

3. What value is returned after executing the following statement? SELECT TO_NUMBER(1234.49, '999999.9') FROM DUAL; (Choose the best answer.)
A. 1234.49
B. 001234.5
C. 1234.5
D. None of the above

4. What value is returned after executing the following statement?
SELECT TO_CHAR(1234.49, '999999.9') FROM DUAL; (Choose the best answer.)
A. 1234.49
B. 001234.5
C. 1234.5
D. None of the above

5. If SYSDATE returns 12-JUL-2009, what is returned by the following statement?
SELECT TO_CHAR(SYSDATE, 'fmMONTH, YEAR') FROM DUAL; (Choose the best answer.)
A. JUL, 2009
B. JULY, TWO THOUSAND NINE
C. JUL-09
D. None of the above

6. If SYSDATE returns 12-JUL-2009, what is returned by the following statement?
SELECT TO_CHAR(SYSDATE, 'fmDDth MONTH') FROM DUAL; (Choose the best answer.)
A. 12TH JULY
B. 12th July
C. TWELFTH JULY
D. None of the above

Apply Conditional Expressions in a SELECT Statement

7. If SYSDATE returns 12-JUL-2009, what is returned by the following statement?
SELECT TO_CHAR(TO_DATE(TO_CHAR(SYSDATE,'DD'),'DD'),'YEAR') FROM DUAL; (Choose the best answer.)
A. 2009
B. TWO THOUSAND NINE
C. 12-JUL-2009
D. None of the above

8. What value is returned after executing the following statement?
SELECT NVL2(NULLIF('CODA','SID'),'SPANIEL','TERRIER') FROM DUAL; (Choose the best answer.)
A. SPANIEL
B. TERRIER
C. NULL
D. None of the above

9. What value is returned after executing the following statement?
SELECT NVL(SUBSTR('AM I NULL',10),'YES I AM') FROM DUAL; (Choose the best answer.)
A. NO
B. NULL
C. YES I AM
D. None of the above

10. If SYSDATE returns 12-JUL-2009, what is returned by the following statement?
SELECT DECODE(TO_CHAR(SYSDATE,'MM'),'02','TAX DUE','PARTY') FROM DUAL; (Choose the best answer.)
A. TAX DUE
B. PARTY
C. 02
D. None of the above

LAB QUESTION

Using SQL Developer or SQL*Plus, connect to the OE schema and complete the following tasks. As part of a new marketing initiative, you are asked to prepare a list of customer birthdays that occur between two days ago and seven days from now. The list should retrieve rows from the CUSTOMERS table which include the CUST_FIRST_NAME, CUST_LAST_NAME, CUST_EMAIL, and DATE_OF_BIRTH columns in ascending order based on the day and month components of the DATE_OF_BIRTH value. An additional expression aliased as BIRTHDAY is required to return a descriptive message based on the following table. There are several approaches to solving this question. Your approach may differ from the solution described here.

Two days ago Day before yesterday
One day ago Yesterday
Today Today
Tomorrow Tomorrow
Two days in the future Day after tomorrow
Within seven days from today Later this week

SELF TEST ANSWERS

Describe Various Types of Conversion Functions Available in SQL

1. ✹✓ B. The number 3.14285 is given as a parameter to the LENGTH function. There is a data type mismatch, but Oracle implicitly converts the parameter to the character string '3.14285', allowing the function to operate correctly.
✹⋎ **A, C,** and **D** are incorrect. Explicit conversion occurs when a function like TO_CHAR is executed. **C** is the correct length of the string '3.14285', but this is not asked for in the question.

2. ✹✓ D. Dates are only converted into character strings using TO_CHAR and not the TO_DATE function.
✹⋎ **A, B,** and **C** are correct statements.

Use the TO_CHAR, TO_NUMBER, and TO_DATE Conversion Functions

3. ✹✓ D. An “ORA-1722: invalid number” error is returned because the statement is trying to convert a number using an incompatible format mask. If the expression was TO_NUMBER(1234.49, '999999.99'), the number 1234.49 would be returned.
✹⋎ **A, B,** and **D** are incorrect.

4. ✹✓ C. For the number 1234.49 to match the character format mask with one decimal place, the number is first rounded to 1234.5 before TO_CHAR converts it into the string '1234.5'.
✹⋎ **A, B,** and **D** are incorrect. **A** cannot be returned because the format mask only allows one character after the decimal point. **B** would be returned if the format mask was '009999.9'.

5. ✹✓ B. The MONTH and YEAR components of the format mask separated by a comma and a space indicate that TO_CHAR must extract the spelled out month and year values in uppercase separated by a comma and a space. The *fm* modifier removes extra blanks from the spelled out components.
✹⋎ **A, C,** and **D** are incorrect. If the format mask was 'MON, YYYY' or 'MON-YY', **A** and **C,** respectively, would be returned.

6. ✹✓ A. The DD component returns the day of the month in uppercase. Since it is a number, it does not matter, unless the 'th' mask is applied, in which case that component is specified in uppercase. MONTH returns the month spelled out in uppercase.
✹⋎ **B, C,** and **D** are incorrect. **B** would be returned if the format mask was 'fmddth Month', and **C** would be returned if the format mask was 'fmDDspth MONTH'.

Apply Conditional Expressions in a SELECT Statement

7. ✹✓ B. The innermost nested function is TO_CHAR(SYSDATE,'DD'), which extracts the day component of SYSDATE and returns the character 12. The next function executed is TO_DATE('12','DD') where the character 12 is cast as the day component. When such an incomplete date is provided, Oracle substitutes values from the SYSDATE function; since SYSDATE is 12-JUL-2009, this is the date used. The outermost function executed in TO_CHAR('12-JUL-2009','YEAR') returns the year spelled out as TWO THOUSAND NINE.
✹⋎ **A, C,** and **D** are incorrect.

8. ✹✓ A. The NULLIF function compares its two parameters and, since they are different, the first parameter is returned. The NVL2('CODA', 'SPANIEL','TERRIER') function call returns SPANIEL since its first parameter is not null.
✹⋎ **B, C,** and **D** are incorrect.

9. ✹✓ C. The character literal 'AM I NULL' is nine characters long. Therefore, trying to obtain a substring beginning at the tenth character returns a null. The outer function then becomes NVL(NULL,'YES I AM'), resulting in the string ‘YES I AM’ being returned.
✹⋎ **A, B,** and **D** are incorrect.

10. ✹✓ **B.** The innermost function TO_CHAR(SYSDATE, 'MM') results in the character string '07' being returned. The outer function is DECODE('07','02','TAX DUE','PARTY'). Since '07' is not equal to '02', the else component 'PARTY' is returned.
✹✗ **A, C,** and **D** are incorrect. **A** would only be returned if the month component extracted from SYSDATE was '02'.

LAB ANSWER

```
SELECT CUST_FIRST_NAME, CUST_LAST_NAME, CUST_EMAIL, DATE_OF_BIRTH,
CASE TO_NUMBER(TO_CHAR(DATE_OF_BIRTH,'DD')) - TO_NUMBER(TO_CHAR(SYSDATE,'DD'))
WHEN -2 THEN 'Day before yesterday'
WHEN -1 THEN 'Yesterday'
WHEN 0 THEN 'Today'
WHEN 1 THEN 'Tomorrow'
WHEN 2 THEN 'Day after tomorrow'
ELSE 'Later this week'
END BIRTHDAY
FROM CUSTOMERS
ORDER BY TO_CHAR(DATE_OF_BIRTH,'MMDD')
```

Chapter 5: Reporting Aggregated Data Using the Group Functions

Single-row functions, return a single value for each row in a set of results. *Group* or *aggregate* functions operate on multiple rows. They are used to count the number of rows or to find the average of specific column values in a dataset.
Many statistical operations, such as calculating standard deviation, medians, and averages, depend on executing functions against grouped data and not just single rows.
The concept of grouping or segregating data based on one or more column values is explored before introducing the *GROUP BY* clause.
The WHERE clause restricts rows in a dataset before grouping, while the *HAVING* clause restricts them after grouping.

Definition of Group Functions
Group functions operate on aggregated data and return a single result per group. These groups usually consist of zero or more rows of data. Single-row functions are defined with the formula: F(x, y, z, …) = result, where x,y,z… are input parameters.
The function F executes on one row of the data set at a time and returns a result for each row. Group functions may be defined using the following formula:
F(g1, g2, g3,…, gn) = result1, result2, result2,…, resultn;
The group function executes once for each cluster of rows and returns a single result per group. These groups may be entire tables or portions of tables associated using a common value or attribute. If all the rows in tables are presented as one group to the group function then one result is returned.
One or more group functions may appear in the SELECT list as follows:

```
SELECT group_function(column or expression),…
FROM table [WHERE …] [ORDER BY…]
```

Consider the EMPLOYEES table. There are 107 rows in this table. Groups may be created based on the common values that rows share. For example, the rows that share the same DEPARTMENT_ID value may be clustered together. Thereafter, group functions are executed separately against each unique group.
Suppose there are 12 distinct DEPARTMENT_ID values in the EMPLOYEES table including a null value. The rows are distributed into 12 groups based on common DEPARTMENT_ID values. The *COUNT* function executes 12 times, once for each group.
Types and Syntax of Group Functions
A brief description of the most commonly used group functions is provided next.

The *COUNT* function counts the number of rows in a group. Its syntax is as follows:
COUNT({*|[DISTINCT|ALL] *expr*}) ;
This syntax may be deconstructed into the following forms:
1. COUNT(*)
2. COUNT(DISTINCT *expr*)
3. COUNT(ALL *expr*)
4. COUNT(*expr*)
When COUNT(*) is invoked, all rows in the group, including those with nulls or duplicate values are counted.
When COUNT(DISTINCT *expr*) is executed, only unique occurrences of *expr* are counted for each group.
The ALL keyword is part of the default syntax, so COUNT(ALL *expr*) and COUNT(*expr*) are equivalent. These count the number of nonnull occurrences of *expr* in each group. The data type of *expr* may be NUMBER, DATE, CHAR, or VARCHAR2. If *expr* is a null, it is ignored unless it is managed using a general function like NVL, NVL2, or COALESCE.

The *AVG* function calculates the average value of a numeric column or expression in a group.
Its syntax is as follows:
AVG([DISTINCT|ALL] *expr*) ;
This syntax may be deconstructed into the following forms:
1. AVG(DISTINCT *expr*)
2. AVG(ALL *expr*)
3. AVG(*expr*)
When AVG(DISTINCT *expr*) is invoked, the distinct values of *expr* are summed and divided by the number of unique occurrences of *expr*.
AVG(ALL *expr*) and AVG(*expr*) add the nonnull values of *expr* for each row and divide the sum by the number of nonnull rows in the group. The data type of the *expr* parameter is NUMBER.

The *SUM* function returns the aggregated total of the nonnull numeric expression values in a group.
It has the following syntax:
SUM([DISTINCT|ALL] *expr*) ;
This syntax may be deconstructed into the following forms:
1. SUM(DISTINCT *expr*)
2. SUM(ALL *expr*)
3. SUM(*expr*)
SUM(DISTINCT *expr*) provides a total by adding all the unique values returned after *expr* is evaluated for each row in the group.
SUM(*expr*) and SUM(ALL *expr*) provide a total by adding *expr* for each row in the group. Null values are ignored. The data type of *expr* is NUMBER.

The *MAX* and *MIN* functions return the maximum (largest) and minimum (smallest) *expr* value in a group.
Their syntax is as follows:
MAX([DISTINCT|ALL] *expr*); MIN([DISTINCT|ALL] *expr*)
This syntax may be deconstructed into the following forms:
1. MAX(DISTINCT *expr*); MIN(DISTINCT *expr*)
2. MAX(ALL *expr*); MIN(ALL *expr*)
3. MAX(*expr*); MIN(*expr*);
MAX(*expr*), MAX(ALL *expr*) and MAX(DISTINCT *expr*) examine the values for *expr* in a group of rows and return the largest value. Null values are ignored.
MIN(*expr*), MIN(ALL *expr*) and MIN(DISTINCT *expr*) examine the values for *expr* in a group of rows and return the smallest value. The data type of the *expr* parameter may be NUMBER, DATE, CHAR or VARCHAR2.

The *STDDEV* and *VARIANCE* functions are two of many statistical group functions Oracle provides.
VARIANCE has the following syntax:
VARIANCE([DISTINCT|ALL] *expr*);
This syntax may be deconstructed into the following forms:
1. VARIANCE(DISTINCT *expr*)
2. VARIANCE(ALL *expr*)

3. VARIANCE(*expr*)

STDDEV has the following syntax:
STDDEV([DISTINCT|ALL] *expr*);
This syntax may be deconstructed into the following forms:
1. STDDEV(DISTINCT *expr*)
2. STDDEV(ALL *expr*)
3. STDDEV(*expr*)
Statistical variance refers to the variability of scores in a sample or set of data.
VARIANCE(DISTINCT *expr*) returns the variability of unique nonnull data in a group.
VARIANCE(*expr*) and VARIANCE(ALL *expr*) return the variability of nonnull data in the group.
STDDEV calculates statistical standard deviation, which is the degree of deviation from the mean value in a group. It is derived by finding the square root of the variance.
STDDEV(DISTINCT *expr*) returns the standard deviation of unique nonnull data in a group.
STDDEV(*expr*) and STDDEV(ALL *expr)* return the standard deviation of nonnull data in the group. The data type of the *expr* parameter is NUMBER.

There are two fundamental rules to remember when studying group functions. First, they always operate on a single group of rows at a time. The group may be one of many groups a dataset has been segmented into or it may be an entire table. The group function executes once per group. Second, rows with nulls occurring in group columns or expressions are ignored, unless a general function like NVL, NVL2, or COALESCE is provided to handle them.

Using the Group Functions
The COUNT Function
The COUNT function has the following syntax:
COUNT({*|[DISTINCT|ALL] *expr*});
There is one parameter that can be either *, which represents all columns including null values, or a specific column or expression. It may be preceded by the DISTINCT or ALL keywords.

```
Query 1: select count(*) from employees
Query 2: select count(commission_pct) from employees
Query 3: select count(distinct commission_pct) from employees
Query 4: select count(hire_date), count(manager_id) from employees
```

Query 1 counts the rows in the EMPLOYEES table and returns the integer 107.
Query 2 counts the rows with nonnull COMMISSION_PCT values and returns 35.
Query 3 considers the 35 nonnull rows, determines the number of unique values, and returns 7.
Query 4 The COUNT function is used on both a DATE column and a NUMBER column. The integers 107 and 106 are returned since there are 107 nonnull HIRE_DATE values and 106 nonnull MANAGER_ID values in the group.

The SUM Function
The aggregated total of a column or an expression is computed with the SUM function.
Its syntax is as follows:
SUM([DISTINCT|ALL] *expr*);
One numeric parameter, optionally preceded by the DISTINCT or ALL keywords, is provided to the SUM function, which returns a numeric value.

```
Query 1: select sum(2) from employees
Query 2: select sum(salary) from employees
Query 3: select sum(distinct salary) from employees
Query 4: select sum(commission_pct) from employees
```

There are 107 rows in the EMPLOYEES table. Query 1 adds the number 2 across 107 rows and returns 214.
Query 2 takes the SALARY column value for every row in the group, which in this case is the entire table, and returns the total salary amount of 691400.
Query 3 returns a total of 397900 since many employees get paid the same salary and the DISTINCT keyword only adds unique values in the column to the total.
Query 4 returns 7.8 after adding the nonnull values.

The AVG Function
The *average* value of a column or expression divides the sum by the number of nonnull rows in the group. The AVG function has the following syntax:
AVG([DISTINCT|ALL] *expr*);
One numeric parameter, preceded by the DISTINCT or ALL keywords, is provided to the AVG function, which returns a numeric value.

```
Query 1: select avg(2) from employees
Query 2: select avg(salary) from employees
Query 3: select avg(distinct salary) from employees
Query 4: select avg(commission_pct) from employees
```

There are 107 rows in the EMPLOYEES table. Query 1 adds the number 2 across 107 rows and divides the total by the number of rows to return the number 2.
Query 2 adds the SALARY value for each row to obtain the total salary amount of 691400. This is divided by the 107 rows with nonnull SALARY values to return the average 6461.68224.
Query 3 There are 57 unique salary values, which when added, yields a total of 397900. Dividing 397900 by 57 returns 6980.70175 as the average of the distinct salary values.
Query 4 may produce unanticipated results if not properly understood. Adding the nonnull values, including duplicates, produces a total of 7.8. There are 35 employee records with nonnull COMMISSION_PCT values. Dividing 7.8 by 35 yields an average COMMISSION_PCT of 0.222857143.

The MAX and MIN Functions
The MAX and MIN functions operate on NUMBER, DATE, CHAR, and VARCHAR2 data types. They return a value of the same data type as their input arguments, which are either the largest or smallest items in the group. When applied to DATE items, MAX returns the latest date and MIN returns the earliest one.

Character strings are converted to numeric representations of their constituent characters based on the NLS settings in the database. When the MIN function is applied to a group of character strings, the word that appears first alphabetically is returned, while MAX returns the word that would appear last.

The MAX and MIN functions have the following syntax:
MAX([DISTINCT|ALL] *expr*);
MIN([DISTINCT|ALL] *expr*)
They take one parameter preceded by the DISTINCT or ALL keywords.

```
Query 1: select min(commission_pct), max(commission_pct) from employees
Query 2: select min(start_date),max(end_date) from job_history
Query 3: select min(job_id),max(job_id) from employees
```

Query 1 returns the numeric values 0.1 and 0.4 for the minimum and maximum COMMISSION_PCT values in the EMPLOYEES table. Note that null values for COMMISSION_PCT are ignored.
Query 2 evaluates a DATE column and indicates that the earliest START_DATE in the JOB_HISTORY table is 17-SEP-1987 and the latest END_DATE is 31-DEC-1999.
Query 3 returns AC_ACCOUNT and ST_MAN as the JOB_ID values appearing first and last alphabetically in the EMPLOYEES table.

Using the Group Functions
The COUNTRIES table stores a list of COUNTRY_NAME values. You are required to calculate the average length of all the country names. Any fractional components must be rounded to the nearest whole number.
SELECT ROUND(AVG(LENGTH(COUNTRY_NAME))) AVERAGE_COUNTRY_NAME_LENGTH
FROM COUNTRIES

Nested Group Functions
Recall that single-row functions may be nested or embedded to any level of depth. *Group functions may only be nested two levels deep*. Three formats using group functions are shown here:
G1(*group_item*) = result
G1(G2(*group_item*) = result
G1(G2(G3(*group_item*))) is NOT allowed.
Group functions are represented by the letter G followed by a number. The first simple form contains no nested functions. Examples include the SUM(*group_item*) or AVG(*group_item*) functions that return a single result per group. The second form supports two nested group functions, like SUM(AVG(*group_item*)). In this case, a GROUP BY clause is mandatory since the average value of the *group_item* per group is calculated before being aggregated by the SUM function.
The third form is disallowed by Oracle. Consider an expression that nests three group functions. If the MAX function is applied to the previous example, the expression MAX(SUM(AVG(*group_item*))) is formed. The two inner group functions return a *single value* representing the sum of a set of average values. This expression becomes MAX(*single value*). A group function cannot be applied to a single value.

Single-row functions may be nested to any level, but group functions may be nested to, at most, two levels deep. The nested function call COUNT(SUM(AVG(X))) returns the error, "ORA-00935: group function is nested too deeply." It is acceptable to nest single-row functions within group functions. Consider the following query:
SELECT SUM(AVG(LENGTH(LAST_NAME))) FROM EMPLOYEES GROUP BY DEPARTMENT_ID.
It calculates the sum of the average length of LAST_NAME values per department.

Creating Groups of Data
A table has at least one column and zero or more rows of data. In many tables this data requires analysis to transform it into useful information. It is a common reporting requirement to calculate statistics from a set of data divided into groups using different attributes. Previous examples using group functions operated against all the rows in a table. The entire table was treated as one large group.
Groups of data within a set are created by associating rows with common properties or attributes with each other. Thereafter, group functions can execute against each of these groups. Groups of data include entire rows and not specific columns.

Consider the EMPLOYEES table. It comprises 11 columns and 107 rows. You could create groups of rows that share a common DEPARTMENT_ID value. The SUM function may then be used to create salary totals per department. Another possible set of groups may share common JOB_ID column values. The AVG group function may then be used to identify the average salary paid to employees in different jobs.

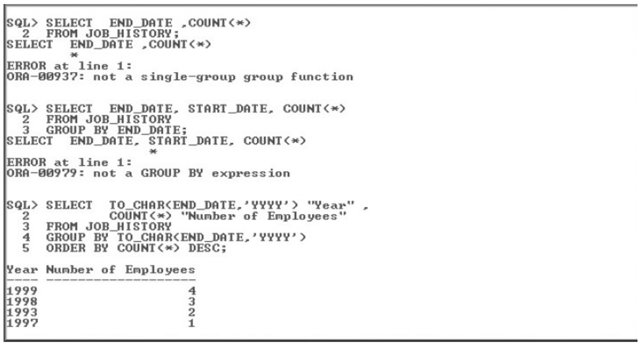
A group is defined as a subset of the entire dataset sharing one or more common attributes. These attributes are typically column values but may also be expressions. The number of groups created depends on the distinct values present in the common attribute.

The GROUP BY Clause
This clause facilitates the creation of groups. It appears after the WHERE clause but before the ORDER BY clause:
SELECT *column*|*expression*|*group_function*(*column*|*expression* [*alias*]),…}
FROM *table*
[WHERE *condition(s)*]
[GROUP BY {*col(s)*|*expr*}]
[ORDER BY {*col(s)*|*expr*|*numeric_pos*} [ASC|DESC] [NULLS FIRST|LAST]];
The column or expression specified in the GROUP BY clause is also known as the *grouping attribute* and is the component that rows are grouped by.
The dataset is segmented based on the grouping attribute. Consider the following query:

```
select max(salary), count(*)
from employees
group by department_id
order by department_id
```

The grouping attribute in this example is the DEPARTMENT_ID column. The dataset, on which the group functions in the SELECT list must operate, is divided into 12 groups, one for each department. For each group (department), the maximum salary value and the number of rows are returned. Since the results are sorted by DEPARTMENT_ID, the third row in the set of results contains the values 11000 and 6. This indicates that 6 employees have a DEPARTMENT_ID value of 30. Of these 6, the highest earner has a SALARY value of 11000. This query demonstrates that the grouping attribute does not have to be included in the SELECT list.

It is common to see the grouping attribute in the SELECT list alongside grouping functions. If an item, which is not a group function, appears in the SELECT list and there is no GROUP BY clause, an "ORA-00937: not a single-group group function" error is raised. If a GROUP BY clause is present but that item is not a grouping attribute, then an "ORA-00979: not a GROUP BY expression" error is returned.
Any item in the SELECT list that is not a group function must be a grouping attribute of the GROUP BY clause. If a group function is placed in a WHERE clause, an "ORA-00934: group function is not allowed here" error is returned. Imposing group-level conditions is achieved using the HAVING clause discussed in the next section. Group functions
may, however, be used as part of the ORDER BY clause.



```
SQL> SELECT  END_DATE ,COUNT(*)
  2  FROM JOB_HISTORY;
SELECT  END_DATE ,COUNT(*)
        *
ERROR at line 1:
ORA-00937: not a single-group group function


SQL> SELECT  END_DATE, START_DATE, COUNT(*)
  2  FROM JOB_HISTORY
  3  GROUP BY END_DATE;
SELECT  END_DATE, START_DATE, COUNT(*)
                  *
ERROR at line 1:
ORA-00979: not a GROUP BY expression


SQL> SELECT  TO_CHAR(END_DATE,'YYYY') "Year" ,
  2          COUNT(*) "Number of Employees"
  3  FROM JOB_HISTORY
  4  GROUP BY TO_CHAR(END_DATE,'YYYY')
  5  ORDER BY COUNT(*) DESC;

Year Number of Employees
---- -------------------
1999                   4
1998                   3
1993                   2
1997                   1
```

The first query raises an error since the END_DATE column is in the SELECT list with a group function and there is no GROUP BY clause.

An “ORA-00979” error is returned from the second query since the START_DATE item is listed in the SELECT clause, but it is not a grouping attribute.

The third query divides the JOB_HISTORY rows into groups based on the 4-digit year component from the END_DATE column. Four groups are created using this grouping attribute. These represent different years when employees ended their jobs. The COUNT shows the number of employees who quit their jobs during each of these years. The results are listed in descending order based on the “Number of Employees” expression. Note that the COUNT group function is present in the ORDER BY clause.

Grouping by Multiple Columns

A powerful extension to the GROUP BY clause uses multiple grouping attributes. Oracle permits datasets to be partitioned into groups and allows these groups to be further divided into subgroups using a different grouping attribute.

```
Query 1: select department_id, sum(commission_pct) from employees
where commission_pct is not null group by department_id
Query 2: select department_id, job_id, sum(commission_pct) from
employees
where commission_pct is not null group by department_id, job_id
```

Query 1 restricts the rows returned from the EMPLOYEES table to the 35 rows ith nonnull COMMISSION_PCT values. These rows are then divided into two groups: 80 and NULL based on the DEPARTMENT_ID grouping attribute. The result set contains two rows, which return the sum of the COMMISSION_PCT values for each group.

Query 2 is similar to the first one except it has an additional item: JOB_ID in both the SELECT and GROUP BY clauses. This second grouping attribute decomposes the two groups based on DEPARTMENT_ID into the constituent JOB_ID components belonging to the rows in each group. The distinct JOB_ID values for rows with DEPARTMENT_ID=80 are SA_REP and SA_MAN. The distinct JOB_ID value for rows with null DEPARTMENT_ID is SA_REP. Therefore, query 2 returns two groupings, one which consists of two subgroups, and the other with only one.

A dataset is divided into groups using the GROUP BY clause. The grouping attribute is the common key shared by members of each group. The grouping attribute is usually a single column but may be multiple columns or an expression that cannot be based on group functions. Note that only grouping attributes and group functions are permitted in the SELECT clause when using GROUP BY.

Grouping Data Based on Multiple Columns

Analysis of staff turnover is a common reporting requirement. You are required to create a report containing the number of employees who left their jobs, grouped by the year in which they left. The jobs they performed are also required. The results must be sorted in descending order based on the number of employees in each group.

The report must list the year, the JOB_ID, and the number of employees who left a particular job in that year.

SELECT TO_CHAR(END_DATE,‘YYYY’) “Quitting Year”, JOB_ID, COUNT(*) “Number of Employees”
FROM EMPLOYEES
GROUP BY TO_CHAR(END_DATE,‘YYYY’), JOB_ID
ORDER BY COUNT(*) DESC

Include or Exclude Grouped Rows Using the HAVING Clause

Creating groups of data and applying aggregate functions is very useful. A refinement to these features is the ability to include or exclude results based on group-level conditions.

Restricting Group Results

WHERE clause conditions restrict rows returned by a query. Rows are included based on whether they fulfill the conditions listed and are sometimes known as *row-level results*. Clustering rows using the GROUP BY clause and applying an aggregate function to these groups returns results often referred to as *group-level results*. The HAVING clause provides the language to restrict group-level results.

The following query limits the rows retrieved from the JOB_HISTORY table by specifying a WHERE condition based on the DEPARTMENT_ID column values.

```
select department_id
from job_history
where department_id in (50,60,80,110);
```

This query returns seven rows. If the WHERE clause was absent, all ten rows would be retrieved. Suppose you want to know how many employees were previously employed in each of these departments. There are seven rows that can be manually grouped and counted. However, if there are a large number of rows, an aggregate function like COUNT may be used, as shown in the following query:

```
select department_id, count(*)
from job_history
where department_id in (50,60,80,110)
group by department_id;
```

This query is very similar to the previous statement. The aggregate function COUNT was added to the SELECT list, and a GROUP BY DEPARTMENT_ID clause was also added. Four rows with their aggregate row count are returned and it is clear that the original seven rows restricted by the WHERE clause were clustered into four groups based on common DEPARTMENT_ID values, as shown below

```
60     1
80     2
110    2
```

Suppose you wanted to refine this list to include only those departments with more than one employee. The HAVING clause limits or restricts the group-level rows as required.

This query must perform the following steps:

1. Consider the entire row-level dataset.
2. Limit the dataset based on any WHERE clause conditions.
3. Segment the data into one or more groups using the grouping attributes specified in the GROUP BY clause.
4. Apply any aggregate functions to create a new group-level dataset. Each row may be regarded as an aggregation of its source row-level data based on the groups created.
5. Limit or restrict the group-level data with a HAVING clause condition. Only group-level results matching these conditions are returned.

The HAVING Clause

The general form of the SELECT statement is further enhanced by the addition of the HAVING clause and becomes:

SELECT *column*|*expression*|*group_function*(*column*|*expression* [*alias*]),…}
FROM *table*
[WHERE *condition(s)*]
[GROUP BY {*col(s)*|*expr*}]
[HAVING *group_condition(s)*]
[ORDER BY {*col(s)*|*expr*|*numeric_pos*} [ASC|DESC] [NULLS FIRST|LAST]];

An important difference between the HAVING clause and the other SELECT statement clauses is that it may only be specified if a GROUP BY clause is present. This dependency is sensible since group-level rows must exist before they can be restricted. The HAVING clause can occur before the GROUP BY clause in the SELECT statement. However, it is more common to place the HAVING clause after the GROUP BY clause. All grouping is performed and group functions are executed prior to evaluating the HAVING clause.

The following query shows how the HAVING clause is used to restrict an aggregated dataset. Records from the JOB_HISTORY table are divided into four groups. The rows that meet the HAVING clause condition (contributing more than one row to the group row count) are returned:

```
select department_id, count(*)
from job_history
where department_id in (50,60,80,110)
group by department_id
having count(*)>1
```

Three rows with DEPARTMENT_ID values of 50, 80, and 110, each with a COUNT(*) value of 2, are returned.

The HAVING clause may only be specified when a GROUP BY clause is present. A GROUP BY clause can be specified without a HAVING clause. Multiple conditions may be imposed by a HAVING clause using the Boolean AND, OR, and NOT operators. The HAVING clause conditions restrict group-level data and must contain a group function or an expression that uses one.

TWO-MINUTE DRILL

Describe the Group Functions

❑ Group functions are also known as multiple row, aggregate, or summary functions. They execute once for each group of data and aggregate the data from multiple rows into a single result for each group.

❑ Groups may be entire tables or portions of a table grouped together by a common grouping attribute.

Identify the Available Group Functions

❑ The COUNT of a function returns an integer value representing the number of rows in a group.

❑ The SUM function returns an aggregated total of all the nonnull numeric expression values in a group.

❑ The AVG function divides the sum of a column or expression by the number of nonnull rows in a group.

❑ The MAX and MIN functions operate on NUMBER, DATE, CHAR, and VARCHAR2 data types. They return a value that is either the largest or smallest item in the group.

❑ Group functions may only be nested two levels deep.

Group Data Using the GROUP BY Clause

❑ The GROUP BY clause specifies the grouping attribute rows must have in common for them to be clustered together.

❑ The GROUP BY clause facilitates the creation of groups within a selected set of data and appears after the WHERE clause but before the ORDER BY clause.

❑ Any item on the SELECT list that is not a group function must be a grouping attribute.

❑ Group functions may not be placed in a WHERE clause.

❑ Datasets may be partitioned into groups and further divided into subgroups based on multiple grouping attributes.

Include or Exclude Grouped Rows Using the HAVING Clause

❑ Clustering rows using a common grouping attribute with the GROUP BY clause and applying an aggregate function to each of these groups returns *group-level results*.

❑ The HAVING clause provides the language to limit the group-level results returned.

❑ The HAVING clause may only be specified if there is a GROUP BY clause present.

❑ All grouping is performed and group functions are executed prior to evaluating the HAVING clause.

SELF TEST

Choose all the correct answers for each question.

Describe the Group Functions

1. What result is returned by the following statement? SELECT COUNT(*) FROM DUAL; (Choose the best answer.)

A. NULL
B. 0

C. 1
D. None of the above
2. Choose one correct statement regarding group functions.
A. Group functions may only be used when a GROUP BY clause is present.
B. Group functions can operate on multiple rows at a time.
C. Group functions only operate on a single row at a time.
D. Group functions can execute multiple times within a single group.

Identify the Available Group Functions

3. What value is returned after executing the following statement?
SELECT SUM(SALARY) FROM EMPLOYEES;
Assume there are 10 employee records and each contains a SALARY value of 100, except for 1, which has a null value in the SALARY field. (Choose the best answer.)
A. 900
B. 1000
C. NULL
D. None of the above
4. Which values are returned after executing the following statement?
SELECT COUNT(*), COUNT(SALARY) FROM EMPLOYEES;
Assume there are 10 employee records and each contains a SALARY value of 100, except for 1, which has a null value in their SALARY field. (Choose all that apply.)
A. 10 and 10
B. 10 and NULL
C. 10 and 9
D. None of the above
5. What value is returned after executing the following statement?
SELECT AVG(NVL(SALARY,100)) FROM EMPLOYEES;
Assume there are ten employee records and each contains a SALARY value of 100, except for one employee, who has a null value in the SALARY field. (Choose the best answer.)
A. NULL
B. 90
C. 100
D. None of the above

Group Data Using the GROUP BY Clause

6. What value is returned after executing the following statement?
SELECT SUM((AVG(LENGTH(NVL(SALARY,0)))))
FROM EMPLOYEES
GROUP BY SALARY;
Assume there are ten employee records and each contains a SALARY value of 100, except for one, which has a null value in the SALARY field. (Choose the best answer.)
A. An error is returned
B. 3
C. 4
D. None of the above
7. How many records are returned by the following query?
SELECT SUM(SALARY), DEPARTMENT_ID FROM EMPLOYEES
GROUP BY DEPARTMENT_ID;
Assume there are 11 nonnull and 1 null unique DEPARTMENT_ID values. All records have a nonnull SALARY value. (Choose the best answer.)
A. 12
B. 11
C. NULL
D. None of the above
8. What values are returned after executing the following statement?
SELECT JOB_ID, MAX_SALARY FROM JOBS GROUP BY MAX_SALARY;
Assume that the JOBS table has ten records with the same JOB_ID value of DBA and the same MAX_SALARY value of 100. (Choose the best answer.)
A. One row of output with the values DBA, 100
B. Ten rows of output with the values DBA, 100
C. An error is returned
D. None of the above

Include or Exclude Grouped Rows Using the HAVING Clause

9. How many rows of data are returned after executing the following statement?
SELECT DEPT_ID, SUM(NVL(SALARY,100)) FROM EMP
GROUP BY DEPT_ID HAVING SUM(SALARY) > 400;
Assume the EMP table has ten rows and each contains a SALARY value of 100, except for one, which has a null value in the SALARY field. The first and second five rows have DEPT_ID values of 10 and 20, respectively. (Choose the best answer.)
A. Two rows
B. One row
C. Zero rows
D. None of the above
10. How many rows of data are returned after executing the following statement?
SELECT DEPT_ID, SUM(SALARY) FROM EMP GROUP BY DEPT_ID HAVING
SUM(NVL(SALARY,100)) > 400;
Assume the EMP table has ten rows and each contains a SALARY value of 100, except for one, which has a null value in the SALARY field. The first and second five rows have DEPT_ID values of 10 and 20, respectively. (Choose the best answer.)
A. Two rows
B. One row
C. Zero rows
D. None of the above

LAB QUESTION

Using SQL Developer or SQL*Plus, connect to the OE schema and complete the following tasks. The PRODUCT_INFORMATION table lists items that are orderable and others that are planned, obsolete, or under development. You are required to prepare a report that groups the nonorderable products by their PRODUCT_STATUS and shows the number of products in each group and the sum of the LIST_PRICE of the products per group. Further, only the group-level rows, where the sum of the LIST_PRICE is greater than 4000, must be displayed. A product is nonorderable if the PRODUCT_STATUS value is not equal to the string ‘orderable’. There are several approaches to solving this question. Your approach may differ from the solution proposed.

SELF TEST ANSWERS

Describe the Group Functions

1. ✹✓ **C.** The DUAL table has one row and one column. The COUNT(*) function returns the number of rows in a table or group.

✹ϒ **A, B,** and **D** are incorrect.

2. ✹✓ **B.** By definition, group functions can operate on multiple rows at a time, unlike single-row functions.

✹ϒ **A, C,** and **D** are incorrect statements. A group function may be used without a GROUP BY clause. In this case, the entire dataset is operated on as a group. The COUNT function is often executed against an entire table, which behaves as one group. **D** is incorrect. Once a dataset has been partitioned into different groups, any group functions execute once per group.

Identify the Available Group Functions

3. ✹✓ **A.** The SUM aggregate function ignores null values and adds nonnull values. Since nine rows contain the SALARY value 100, 900 is returned.

✹ϒ **B, C,** and **D** are incorrect. **B** would be returned if SUM(NVL(SALARY,100)) was executed. **C** is a tempting choice since regular arithmetic with NULL values returns a NULL result. However, the aggregate functions, except for COUNT(*), ignore NULL values.

4. ✹✓ **C.** COUNT(*) considers all rows including those with NULL values. COUNT(SALARY) only considers the nonnull rows.

✹ϒ **A, B,** and **D** are incorrect.

5. ✹✓ **C.** The NVL function converts the one NULL value into 100. Thereafter, the average function adds the SALARY values and obtains 1000. Dividing this by the number of records returns 100.

✹ϒ **A, B,** and **D** are incorrect. **B** would be returned if AVG(NVL(SALARY,0)) was selected. It is interesting to note that if AVG(SALARY) was selected, 100 would have also been returned, since the AVG function would sum the nonnull values and divide the total by the number of rows with nonnull SALARY values. So AVG(SALARY) would be calculated as: 900/9=100.

Group Data Using the GROUP BY Clause

6. ✹✓ **C.** The dataset is segmented based on the SALARY column. This creates two groups: one with SALARY values of 100 and the other with a null SALARY value. The average length of SALARY value 100 is 3 for the rows in the first group. The NULL salary value is first converted into the number 0 by the NVL function, and the average length of SALARY is 1. The SUM function operates across the two groups adding the values 3 and 1 returning 4.

✹ϒ **A, B,** and **D** are incorrect. **A** seems plausible since group functions may not be nested more than two levels deep. Although there are four functions, only two are group functions while the others are single-row functions evaluated before the group functions. **B** would be returned if the expression SUM(AVG(LENGTH(SALARY))) was selected.

7. ✹✓ **A.** There are 12 distinct DEPARTMENT_ID values. Since this is the grouping attribute, 12 groups are created, including 1 with a null DEPARTMENT_ID value. Therefore 12 rows are returned.

✹ϒ **B, C,** and **D** are incorrect.

Include or Exclude Grouped Rows Using the HAVING Clause

8. ✹✓ **C.** For a GROUP BY clause to be used, a group function must appear in the SELECT list.

✹ϒ **A, B,** and **D** are incorrect since the statement is syntactically inaccurate and is disallowed by Oracle. Do not mistake the column named MAX_SALARY for the MAX(SALARY) function.

9. ✹✓ **B.** Two groups are created based on their common DEPT_ID values. The group with DEPT_ID values of ten consists of five rows with SALARY values of 100 in each of them. Therefore, the SUM(SALARY) function returns 500 for this group, and it satisfies the HAVING SUM(SALARY) > 400 clause. The group with DEPT_ID values of 20 has four rows with SALARY values of 100 and one row with a NULL SALARY. SUM(SALARY) only returns 400 and this group does not satisfy the HAVING clause.

✹ϒ **A, C,** and **D** are incorrect. Beware of the SUM(NVL(SALARY,100)) expression in the SELECT clause. This expression selects the format of the output. It does not restrict or limit the dataset in anyway.

10. ✹✓ **A.** Two groups are created based on their common DEPT_ID values. The group with DEPT_ID values of 10 consists of five rows with SALARY values of 100 in each of them. Therefore the SUM(NVL(SALARY,100)) function returns 500 for this group and it satisfies the HAVING SUM(SALARY) > 400 clause. The group with DEPT_ID values of 20 has four rows with SALARY values of 100 and one row with a null SALARY. SUM(NVL(SALARY,100)) returns 500 and this group satisfies the HAVING clause. Therefore two rows are returned.

✹ϒ **B, C,** and **D** are incorrect. Although the SELECT clause contains SUM(SALARY), which returns 500 and 400 for the two groups, the HAVING clause contains the SUM(NVL(SALARY,100)) expression, which specifies the inclusion or exclusion criteria for a group-level row.

LAB ANSWER

```
SELECT COUNT(*), SUM(LIST_PRICE), PRODUCT_STATUS
FROM PRODUCT_INFORMATION
WHERE UPPER(PRODUCT_STATUS) <> 'ORDERABLE'
GROUP BY PRODUCT_STATUS
HAVING SUM(LIST_PRICE) > 4000
```

Chapter 6: Displaying Data from Multiple Tables

Rows from different tables are associated with each other using *joins*. Support for joining has implications for the way data is stored in database tables.

Tables may be joined in several ways.

The most common technique is called an *equijoin*. A row is associated with one or more rows in another table based on the *equality* of column values or expressions.

Tables may also be joined using a *nonequijoin*. In this case, a row is associated with one or more rows in another table if its column values fall into a range determined by inequality operators. A less common technique is to associate rows with other rows in the same table. This association is based on columns with logical and usually hierarchical relationships with each other. This is called a *self-join*. Rows with null or differing entries in common *join columns* are excluded when equijoins and nonequijoins are performed. An *outer join* is available to fetch these *one-legged* or *orphaned* rows if necessary.

A *cross join* or *Cartesian product* is formed when every row from one table is joined to all rows in another. This join is often the result of missing or inadequate join conditions but is occasionally intentional.

Types of Joins

Two basic joins are the *equijoin* and the *nonequijoin*. Equijoins are more frequently used. Joins may be performed between multiple tables. The first table is called the *source* and the second is called the *target.* Rows in the source and target tables comprise one or more columns.

As an example, assume that the *source* and *target* are the COUNTRIES and REGIONS tables from the HR schema, respectively. The COUNTRIES table contains three columns named COUNTRY_ID, COUNTRY_NAME, and REGION_ID. The REGIONS table is comprised of two columns named REGION_ID and REGION_NAME. The data in these two tables is related to each other based on the common REGION_ID column.

```
Query 1: select * from countries where country_id='CA';
Query 2: select region_name from regions where region_id='2';
```

The name of the region to which a country belongs may be determined by obtaining its REGION_ID value. This value is used to join it with the row in the REGIONS table with the same REGION_ID.

Query 1 retrieves the column values associated with the row from the COUNTRIES table where the COUNTRY_ID='CA'. The REGION_ID value of this row is 2.

Query 2 fetches the Americas REGION_NAME from the REGIONS table for the row with REGION_ID=2. Equijoining facilitates the retrieval of column values from multiple tables using a single query.

The source and target tables can be swapped, so the REGIONS table could be the source and the COUNTRIES table could be the target.

```
Query 1: select * from regions where region_name='Americas';
Query 2: select country_name from countries where region_id='2';
```

Query 1 fetches one row with a REGION_ID value of 2. Joining in this reversed manner allows the following question to be asked: What countries belong to the Americas region? The answers from the second query are five COUNTRY_NAME values: Argentina, Brazil, Canada, Mexico, and the United States of America.

These results may be obtained from a single query that joins the tables together. The language to perform equijoins, nonequijoins, outer joins, and cross joins.

Natural Joins

The natural join is implemented using three possible *join clauses* that use the following keywords in different combinations: *NATURAL JOIN, USING,* and *ON.*

When the source and target tables share identically named columns, it is possible to perform a natural join between them without specifying a join column. This is sometimes referred to as a *pure natural join*. In this scenario, columns with the same names in the source and target tables are automatically associated with each other. Rows with matching column values in both tables are retrieved.

The REGIONS and COUNTRIES table both share the REGION_ID column. They may be naturally joined without specifying join columns.

The NATURAL JOIN keywords instruct Oracle to identify columns with identical names between the source and target tables. Thereafter, a join is implicitly performed between them. In the first query, the REGION_ID column is identified as the only commonly named column in both tables. REGIONS is the source table and appears after the FROM clause. The target table is therefore COUNTRIES. For each row in the REGIONS table, a match for the REGION_ID value is sought from all the rows in the COUNTRIES table.

An interim result set is constructed containing rows matching the join condition. This set is then restricted by the WHERE clause.

Sometimes more control must be exercised regarding which columns to use for joins. When there are identical column names in the source and target tables you want to exclude as join columns, the *JOIN…USING* format may be used.

Remember that Oracle does not impose any rules stating that columns with the same name in two discrete tables must have a relationship with each other.

Be wary when using pure natural joins since database designers may assign the same name to key or unique columns. These columns may have names like ID or SEQ_NO. If a pure natural join is attempted between such tables, ambiguous and unexpected results may be returned.

Outer Joins

Not all tables share a perfect relationship, where every record in the source table can be matched to at least one row in the target table. It is occasionally required that rows with nonmatching join column values also be retrieved by a query. This may seem to defeat the purpose of joins but has some practical benefits.

Suppose the EMPLOYEES and DEPARTMENTS tables are joined with common DEPARTMENT_ID values. EMPLOYEES records with null DEPARTMENT_ID values are excluded along with values absent from the DEPARTMENTS table. An *outer join* fetches these rows.

Cross Joins

A *cross join* or *Cartesian product* derives its names from mathematics, where it is also referred to as a cross product between two sets or matrices.

This join creates one row of output for every combination of source and target table rows. If the source and target tables have three and four rows, respectively, a cross join between them results in (3 ∗ 4 = 12) rows being returned.

Oracle Join Syntax

The Oracle join syntax supports natural joining, outer joins, and Cartesian joins, as shown in the following queries:

```
Query 1: select regions.region_name, countries.country_name
from regions, countries
where regions.region_id=countries.region_id;
Query 2: select last_name, department_name
from employees, departments
where employees.department_id (+) = departments.department_id;
```

```
Query 3: select * from regions,countries;
```

Query 1 performs a natural join by specifying the join as a condition in the WHERE clause. Take note of the column aliasing using the TABLE.COLUMN_NAME notation to disambiguate the identical column names.

Query 2 specifies the join between the source and target tables as a WHERE condition. There is a plus symbol enclosed in brackets (+) to the *left* of the equal sign that indicates to Oracle that a *right outer join* must be performed. This query returns employees' LAST_NAME and their matching DEPARTMENT_NAME values. In addition, the outer join retrieves DEPARTMENT_NAME from the rows with DEPARTMENT_ID values not currently assigned to any employee records. Query 3 performs a Cartesian or cross join by excluding the join condition.

Joining Tables Using SQL:1999 Syntax

Prior to Oracle 9*i*, the traditional join syntax was the only language available to join tables. Since then, Oracle has introduced a new language that is compliant to the ANSI SQL:1999 standards. It offers no performance benefits over the traditional syntax. Natural, outer, and cross joins may be written using both SQL:1999 and traditional Oracle SQL.

The general form of the SELECT statement using ANSI SQL:1999 syntax is as follows:

SELECT *table1.column, table2.column*
FROM *table1*
[NATURAL JOIN *table2*] |
[JOIN *table2* USING (*column_name*)] |
[JOIN *table2* ON (*table1.column_name = table2.column_name*)] |
[LEFT | RIGHT | FULL OUTER JOIN *table2*
ON (*table1.column_name = table2.column_name*)] |
[CROSS JOIN *table2*];

The general form of the traditional Oracle-proprietary syntax relevant to joins is as follows:

SELECT *table1.column, table2.column*
FROM *table1, table2*
[WHERE (*table1.column_name = table2.column_name*)] |
[WHERE (*table1.column_name(+)= table2.column_name*)] |
[WHERE (*table1.column_name)= table2.column_name*) (+)] ;

If no joins or fewer than N-1 joins are specified in the WHERE clause conditions, where N refers to the number of tables in the query, then a Cartesian or cross join is performed. If an adequate number of join conditions is specified, then the first optional conditional clause specifies a natural join, while the second two optional clauses specify the syntax for right and left outer joins.

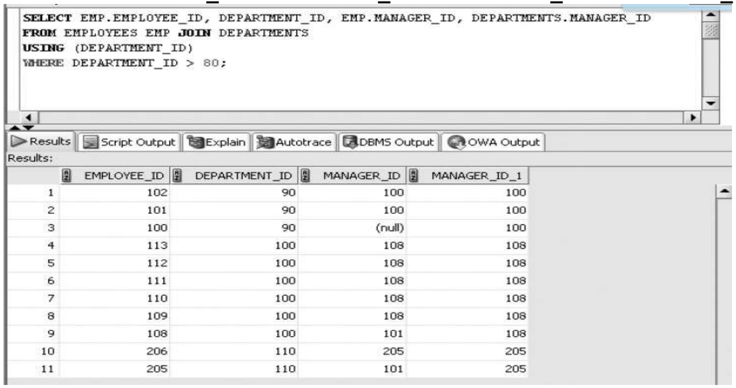
Qualifying Ambiguous Column Names

Columns with the same names may occur in tables involved in a join. The columns named DEPARTMENT_ID and MANAGER_ID are found in both the EMPLOYEES and DEPARTMENTS tables. The REGION_ID column is present in both the REGIONS and COUNTRIES tables. Listing such columns in a query becomes problematic when Oracle cannot resolve their origin. Columns with unique names across the tables involved in a join cause no ambiguity, and Oracle can easily resolve their source table.

The problem of ambiguous column names is addressed with dot notation. A column may be prefixed by its table name and a dot or period symbol to designate its origin. This differentiates it from a column with the same name in another table.

Dot notation may be used in queries involving any number of tables. Referencing some columns using dot notation does not imply that all columns must be referenced in this way.

Dot notation is enhanced with table aliases. A *table alias* provides an alternate, usually shorter name for a table. A column may be referenced as *TABLE_NAME.COLUMN_NAME* or *TABLE_ALIAS.COLUMN_NAME.*



```
SELECT EMP.EMPLOYEE_ID, DEPARTMENT_ID, EMP.MANAGER_ID, DEPARTMENTS.MANAGER_ID
FROM EMPLOYEES EMP JOIN DEPARTMENTS
USING (DEPARTMENT_ID)
WHERE DEPARTMENT_ID > 80;
```

Results:

	EMPLOYEE_ID	DEPARTMENT_ID	MANAGER_ID	MANAGER_ID_1
1	102	90	100	100
2	101	90	100	100
3	100	90	(null)	100
4	113	100	108	108
5	112	100	108	108
6	111	100	108	108
7	110	100	108	108
8	109	100	108	108
9	108	100	101	108
10	206	110	205	205
11	205	110	101	205

The EMPLOYEES table is aliased with the short name EMP while the DEPARTMENTS table is not. The SELECT clause references the EMPLOYEE_ID and MANAGER_ID columns as EMP.EMPLOYEE_ID and EMP.MANAGER_ID.

The MANAGER_ID column from the DEPARTMENTS table is referred to as DEPARTMENTS.MANAGER_ID. Qualifying the EMPLOYEE_ID column using dot notation is unnecessary because there is only one column with this name between the two tables. Therefore, there is no ambiguity. The MANAGER_ID column must be qualified to avoid ambiguity because it is featured in both tables. Since the JOIN…USING format is applied*,* only DEPARTMENT_ID is used as the join column. If a NATURAL JOIN was employed, both the DEPARTMENT_ID and MANAGER_ID columns would be used. If the MANAGER_ID column was not qualified, an "ORA-00918:column ambiguously defined" error would be returned. If DEPARTMENT_ID was aliased, an "ORA-25154:column part of USING clause cannot have qualifier" error would be raised. SQL Developer provides the heading MANAGER_ID to the first reference made in the SELECT clause. The string "_1" is automatically appended to the second reference, creating the heading MANAGER_ID_1.

Qualifying column references with dot notation to indicate a column's table of origin has a performance benefit. Time is saved because Oracle is directed instantaneously to the appropriate table and does not have to resolve the table name.

The NATURAL JOIN Clause

The general syntax for the NATURAL JOIN clause is as follows:

SELECT table1.column, table2.column
FROM table1
NATURAL JOIN table2;

The pure natural join identifies the columns with common names in table1 and table2 and implicitly joins the tables using all these columns. The columns in the SELECT clause may be qualified using dot notation unless they are one of the join columns.

Query 1: *select * from locations natural join countries;*

*Query 2: select * from locations, countries where locations.country_id = countries.country_id;*
*Query 3: select * from jobs natural join countries;*
*Query 4: select * from jobs, countries;*
The natural join identifies columns with common names between the two tables.
In query 1, COUNTRY_ID occurs in both tables and becomes the join column.
Query 2 is written using traditional Oracle syntax and retrieves the same rows as query 1. Unless you are familiar with the columns in the source and target tables, natural joins must be used with caution, as join conditions are automatically formed between all columns with shared names.
Query 3 performs a natural join between the JOBS and COUNTRIES tables. There are no columns with identical names, resulting in a Cartesian product.
Query 4 is equivalent to query 3, and a Cartesian join is performed using traditional Oracle syntax.

The natural join is simple but prone to a fundamental weakness. It suffers the risk that two columns with the same name might have no relationship and may not even have compatible data types.

Using the NATURAL JOIN
The JOB_HISTORY table shares three identically named columns with the EMPLOYEES table: EMPLOYEE_ID, JOB_ID, and DEPARTMENT_ID. You are required to describe the tables and fetch the EMPLOYEE_ID, JOB_ID, DEPARTMENT_ID, LAST_NAME, HIRE_DATE, and END_DATE values for all rows retrieved using a pure natural join. Alias the EMPLOYEES table as EMP and the JOB_HISTORY table as JH and use dot notation where possible.

SELECT EMP.LAST_NAME, EMP.HIRE_DATE, JH.END_DATE
FROM JOB_HISTORY JH NATURAL JOIN EMPLOYEES EMP

The Natural JOIN USING Clause
The format of the syntax for the natural JOIN USING clause is as follows:
SELECT table1.column, table2.column
FROM table1
JOIN table2 USING (join_column1, join_column2…);
While the pure natural join contains the NATURAL keyword in its syntax, the JOIN…USING syntax does not. An error is raised if the keywords NATURAL and USING occur in the same join clause. The JOIN…USING clause allows one or more equijoin columns to be explicitly specified in brackets after the USING keyword. This avoids the shortcomings associated with the pure natural join. Many situations demand that tables be joined only on certain columns, and this format caters to this requirement.

Query 1: select * from locations join countries using (country_id);
Query 2: select * from locations, countries where locations.country_id = countries.country_id;
Query 3: select * from jobs join countries using ;
Query 1 specifies that the LOCATIONS and COUNTRIES tables must be joined on common COUNTRY_ID column values. All columns from these tables are retrieved for the rows with matching join column values.
Query 2 shows a traditionally specified query that retrieves the same rows as query 1.
Query 3 illustrates that a Cartesian join cannot be accidentally specified with the JOIN…USING syntax since only columns with shared names are permitted after the USING keyword. The join columns cannot be qualified using table names or aliases when they are referenced. Since this join syntax potentially excludes some columns with identical names from the join clause, these must be qualified if they are referenced to avoid ambiguity.

The Natural JOIN ON Clause
The format of the syntax for the natural JOIN ON clause is as follows:
SELECT *table1.column, table2.column*
FROM *table1* JOIN *table2* ON (*table1.column_name* = *table2.column_name*);
The pure natural join and the JOIN…USING clauses depend on join columns with identical column names. The JOIN…ON clause allows the explicit specification of join columns, regardless of their column names. This is the most
flexible and widely used form of the join clauses. The ON and NATURAL keywords cannot appear together in a join clause. The equijoin columns are fully qualified as *table1.column1 = table2.column2* and are optionally specified in brackets after the ON keyword.

```
Query 1: select * from departments d join employees e on (e.employee_id=d.department_id);
Query 2: select * from employees e, departments d where e.employee_id=d.department_id;
```

Query 1 retrieves all column values from both the DEPARTMENTS and EMPLOYEES tables for the rows that meet an equijoin condition. This condition is fulfilled by EMPLOYEE_ID values matching DEPARTMENT_ID values in the DEPARTMENTS table.
The traditional Oracle syntax in query 2 returns the same results as query 1. Notice the similarities between the traditional join condition specified in the WHERE clause and the join condition specified after the ON keyword.
The START_DATE column in the JOB_HISTORY table is joined to the HIRE_DATE column in the EMPLOYEES table

Using the NATURAL JOIN…ON Clause
Each record in the DEPARTMENTS table has a MANAGER_ID column matching an EMPLOYEE_ID value in the EMPLOYEES table. You are required to produce a report with one column aliased as Managers. Each row must contain a sentence of the format FIRST_NAME LAST_NAME is manager of the DEPARTMENT_NAME department. Alias the EMPLOYEES table as E and the DEPARTMENTS table as D and use dot notation where possible.

SELECT E.FIRST_NAME||' '||E.LAST_NAME||' is manager of the '||D.DEPARTMENT_NAME||' department.' "Managers"
FROM EMPLOYEES E JOIN DEPARTMENTS D ON (E. EMPLOYEE_ID=D.MANAGER_ID).

N-Way Joins and Additional Join Conditions
The joins just discussed were demonstrated using two tables. There is no restriction on the number of tables that may be related using joins. Third normal form consists of a set of tables connected through a series of primary and foreign key relationships.
Traversing these relationships using joins enables consistent and reliable retrieval of data. When multiple joins exist in a statement, they are evaluated from left to right.
Consider the following query using pure natural joins:

```
select r.region_name, c.country_name, l.city, d.department_name
from departments d natural join locations l
natural join countries c natural join regions r
```

The join between DEPARTMENTS and LOCATIONS creates an interim result set consisting of 27 rows. These tables provide the DEPARTMENT_NAME and CITY columns. This set is naturally joined to the COUNTRIES table. Since the interim set does not contain the COUNTRY_ID column, a Cartesian join is performed. The 27 interim rows are joined to the 25 rows in the COUNTRIES table, yielding a new interim results set with 675 (27 · 25) rows and three columns:

DEPARTMENT_NAME, CITY, and COUNTRY_NAME. This set is naturally joined to the REGIONS table. Once again, a Cartesian join occurs because the REGION_ID column is absent from the interim set. The final result set contains 2700 (675 · 4) rows and four columns. Using pure natural joins with multiple tables is error prone and not recommended.
The JOIN…USING and JOIN…ON syntaxes are better suited for joining multiple tables.

The following query joins four tables using the pure natural join syntax:

```
select region_id, country_id, c.country_name, l.city, d.department_name
from departments d natural join locations l
natural join countries c natural join regions r
```

This query correctly yields 27 rows in the final results set since the required join columns are listed in the SELECT clause. The following query demonstrates how the JOIN…ON clause is used to fetch the same 27 rows. A join condition can reference only columns in its scope. In the following example, the join from DEPARTMENTS to LOCATIONS may not reference columns in the COUNTRIES or REGIONS tables, but the join between COUNTRIES and REGIONS may reference any column from the four tables involved in the query.

```
select r.region_name, c.country_name, l.city, d.department_name
from departments d
join locations l on (l.location_id=d.location_id)
join countries c on (c.country_id=l.country_id)
join regions r on (r.region_id=c.region_id)
```

The JOIN…USING clause can also be used to join these four tables as follows:

```
select r.region_name, c.country_name, l.city, d.department_name
from departments d
join locations l using (location_id)
join countries c using (country_id)
join regions r using (region_id)
```

The WHERE clause is used to specify conditions that restrict the results set of a query whether it contains joins or not. The JOIN…ON clause is also used to specify conditions that limit the results set created by the join.
Consider the following two queries:

```
Query 1: select d.department_name from departments d
join locations l on (l.LOCATION_ID=d.LOCATION_ID)
where d.department_name like 'P%'
Query 2: select d.department_name from departments d
join locations l on
(l.LOCATION_ID=d.LOCATION_ID and d.department_name like 'P%')
```

Query 1 uses a WHERE clause to restrict the 27 rows created by equijoining the DEPARTMENTS and LOCATIONS tables based on their LOCATION_ID values to the three that contain DEPARTMENT_ID values beginning with the letter P.
Query 2 implements the condition within the brackets of the ON subclause and returns the same three rows.

Nonequijoins

Nonequijoins match column values from different tables based on an inequality expression. The value of the join column in each row in the source table is compared to the corresponding values in the target table. A match is found if the expression used in the join, based on an inequality operator, evaluates to true. When such a join is constructed, a nonequijoin is performed.
A nonequijoin is specified using the JOIN…ON syntax, but the join condition contains an inequality operator instead of an equal sign.

The format of the syntax for a nonequijoin clause is as follows:

```
SELECT table1.column, table2.column
FROM table1
[JOIN table2 ON (table1.column_name < table2.column_name)]|
[JOIN table2 ON (table1.column_name > table2.column_name)]|
[JOIN table2 ON (table1.column_name <= table2.column_name)]|
[JOIN table2 ON (table1.column_name >= table2.column_name)]|
[JOIN table2 ON (table1.column BETWEEN table2.col1 AND table2.col2)]|
```

Nonequijoins are not commonly used. The BETWEEN range operator often appears with nonequijoin conditions. It is simpler to use one BETWEEN operator in a condition than two nonequijoin conditions based on the less than or equal to (<=) and the greater than or equal to (>=) operators.

Join a Table to Itself Using a Self-Join

Storing hierarchical data in a single relational table is accomplished by allocating at least two columns per row. One column stores an identifier of the row's parent record and the second stores the row's identifier. Associating rows with each other based on a hierarchical relationship requires Oracle to join a table to itself.

Joining a Table to Itself Using the JOIN…ON Clause

Suppose there is a need to store a family tree in a relational table. There are several approaches one could take. One option is to use a table called FAMILY with columns named ID, NAME, MOTHER_ID, and FATHER_ID, where each row stores a person's name, unique ID number, and the ID values for their parents.
When two tables are joined, each row from the source table is subjected to the join condition with rows from the target table. If the condition evaluates to true, then the joined row, consisting of columns from both tables, is returned.
When the join columns originate from the same table, a self-join is required. Conceptually, the source table is duplicated to create the target table. The self-join works like a regular join between these tables.
Note that, internally, Oracle does not duplicate the table and this description is merely provided to explain the concept of self-joining.
Consider the following four queries:

```
Query 1: select id, name, father_id from family;
Query 2: select name from family where id=&father_id;
Query 3: select f1.name Dad, f2.name Child
from family f1 join family f2 on (f1.id=f2.father_id)
```

To identify a person's father in the FAMILY table, you could use query 1 to get that person's ID, NAME, and FATHER_ID value.
In query 2, the FATHER_ID value obtained from the first query can be substituted to obtain the father's NAME value. Notice that both queries 1 and 2 source information from the FAMILY table.
Query 3 performs a self-join with the JOIN…ON clause by aliasing the FAMILY table as f1 and f2. Oracle treats these as different tables even though they point to the same physical table. The first occurrence of the FAMILY table, aliased as f1, is designated as the source table, while the second occurrence, aliased as f 2, is assigned as the target table. The join condition in the ON clause is of the format *source.child_id=target.parent_id.*

Performing a Self-Join

There is a hierarchical relationship between employees and their managers. For each row in the EMPLOYEES table the MANAGER_ID column stores the EMPLOYEE_ID of every employee's manager. Using a self-join on the EMPLOYEES table, you are required to retrieve the employee's LAST_NAME, EMPLOYEE_ID, manager's LAST_NAME, and employee's DEPARTMENT_ID for the rows with DEPARMENT_ID values of 10, 20, or 30. Alias the EMPLOYEES table as E and the second instance of the EMPLOYEES table as M. Sort the results based on the DEPARTMENT_ID column.

```
SELECT E.LAST_NAME EMPLOYEE, E.EMPLOYEE_ID, E.MANAGER_ID,
M.LAST_NAME MANAGER, E.DEPARTMENT_ID.
FROM EMPLOYEES E JOIN EMPLOYEES M ON (E.MANAGER_ID=M.EMPLOYEE_ID)
WHERE E.DEPARTMENT_ID IN (10,20,30).
ORDER BY E.DEPARTMENT_ID.
```

View Data That Does Not Meet a Join Condition by Using Outer Joins

Equijoins match rows between two tables based on the equality of the column data stored in each table. Nonequijoins rely on matching rows between tables based on a join condition containing an inequality expression. Target table rows with no matching join column in the source table are usually not required. When they are required, however, an *outer join* is used to fetch them. Several variations of outer joins may be used depending on whether join column data is missing from the source or target tables or both. These outer join techniques are described in the following topics:

- Inner versus outer joins
- Left outer joins
- Right outer joins
- Full outer joins

There are three types of outer join formats. Each of them performs an inner join before including rows the join condition excluded. If a left outer join is performed then rows excluded by the inner join, to the left of the JOIN keyword, are also returned. If a right outer join is performed then rows excluded by the inner join, to the right of the JOIN keyword, are returned as well.

Inner versus Outer Joins

When equijoins and nonequijoins are performed, rows from the source and target tables are matched using a join condition formulated with equality and inequality operators, respectively. These are referred to as *inner joins.* An *outer join* is performed when rows, which are not retrieved by an inner join*,* are returned. Two tables sometimes share a *master-detail* or *parent-child* relationship.

In the sample HR schema there are several pairs of tables with such a relationship. One pair is the DEPARTMENTS and EMPLOYEES tables. The DEPARTMENTS table stores a master list of DEPARTMENT_NAME and DEPARTMENT_ID values. Each EMPLOYEES record has a DEPARTMENT_ID column constrained to be either a value that exists in the DEPARTMENTS table or null. This leads to one of the following three scenarios. The fourth scenario could occur if the constraint between the tables was removed.

1. An employee row has a DEPARTMENT_ID value that matches a row in the DEPARTMENTS table.
2. An employee row has a null value in its DEPARTMENT_ID column.
3. There are rows in the DEPARTMENTS table with DEPARTMENT_ID values that are not stored in any employee records.
4. An employee row has a DEPARTMENT_ID value that is not featured in the DEPARTMENTS table.

The first scenario describes a natural inner join between the two tables. The second and third scenarios cause many problems. Joining the EMPLOYEES and DEPARTMENTS tables results in employee rows being excluded. An outer join can be used to include these orphaned rows in the results set. The fourth scenario should rarely occur in a well designed database, because foreign key constraints would prevent the insertion of child records with no parent values. Since this row will be excluded by an inner join, it may be retrieved using an outer join.

A left outer join between the source and target tables returns the results of an inner join as well as rows from the source table excluded by that inner join. A right outer join between the source and target tables returns the results of an inner join as well as rows from the target table excluded by that inner join. If a join returns the results of an inner join as well as rows from both the source and target tables excluded by that inner join, then a full outer join has been performed.

Left Outer Joins

The format of the syntax for the LEFT OUTER JOIN clause is as follow:

```
SELECT table1.column, table2.column
FROM table1
LEFT OUTER JOIN table2
ON (table1.column = table2.column);
```

A left outer join performs an inner join of *table1* and *table2* based on the condition specified after the ON keyword. Any rows from the table on the *left* of the JOIN keyword excluded for not fulfilling the join condition are also returned.

Consider the following two queries:

```
Query 1: select e.employee_id, e.department_id EMP_DEPT_ID,
d.department_id DEPT_DEPT_ID, d.department_name
from departments d left outer join employees e on (d.DEPARTMENT_ID=e.DEPARTMENT_ID)
where d.department_name like 'P%'
Query 2: select e.employee_id, e.department_id EMP_DEPT_ID,
d.department_id DEPT_DEPT_ID, d.department_name
from departments d join employees e on (d.DEPARTMENT_ID=e.DEPARTMENT_ID)
where d.department_name like 'P%'
```

Queries 1 and 2 are identical except for the join clauses, which have the keywords LEFT OUTER JOIN and JOIN*,* respectively.

Query 2 performs an inner join and seven rows are returned. These rows share identical DEPARTMENT_ID values in both tables.

Query 1 returns the same seven rows and one additional row. This extra row is obtained from the table to the left of the JOIN keyword, which is the DEPARTMENTS table. It is the row containing details of the Payroll department. The inner join does not include this row since no employees are currently assigned to the department.

Right Outer Joins

The format of the syntax for the RIGHT OUTER JOIN clause is as follows:

```
SELECT table1.column, table2.column
FROM table1
RIGHT OUTER JOIN table2
ON (table1.column = table2.column);
```

A right outer join performs an inner join of *table1* and *table2* based on the join condition specified after the ON keyword. Rows from the table to the *right* of the JOIN keyword, excluded by the join condition, are also returned.
Consider the following query:

```
select e.last_name, d.department_name from departments d
right outer join employees e on (e.department_id=d.department_id)
where e.last_name like 'G%';
```

The inner join produces seven rows containing details for the employees with LAST_NAME values that begin with G. The EMPLOYEES table is to the *right* of the JOIN keyword. Any employee records which do not conform to the join condition are included, provided they conform to the WHERE clause condition. In addition, the right outer join fetches one EMPLOYEE record with a LAST_NAME of Grant. This record currently has a null DEPARTMENT_ID value. The inner join excludes the record since no DEPARTMENT_ID is assigned to this employee.

Full Outer Joins

The format of the syntax for the FULL OUTER JOIN clause is as follows:

SELECT *table1.column, table2.column*
FROM *table1*
FULL OUTER JOIN *table2*
ON (*table1.column* = *table2.column*);

A *full outer join* returns the combined results of a left and right outer join. An inner join of *table1* and *table2* is performed before rows excluded by the join condition from both tables are merged into the results set. The traditional Oracle join syntax does not support a full outer join, which is typically performed by combining the results from a left and right outer join using the UNION set operator.

Performing an Outer-Join

The DEPARTMENTS table contains details of all departments in the organization. You are required to retrieve the DEPARTMENT_NAME and DEPARTMENT_ID values for those departments to which no employees are currently assigned.

```
SELECT D.DEPARTMENT_NAME, D.DEPARTMENT_ID.
FROM DEPARTMENTS D LEFT OUTER JOIN EMPLOYEES E ON E.DEPARTMENT_ID=D.DEPARTMENT_ID
WHERE E.DEPARTMENT_ID IS NULL.
```

Generate a Cartesian Product of Two or More Tables

A Cartesian product of two tables may be conceptualized as joining each row of the source table with every row in the target table. The number of rows in the result set created by a Cartesian product is equal to the number of rows in the source table multiplied by the number of rows in the target table. Cartesian products may be formed intentionally using the ANSI SQL:1999 cross join syntax.

Creating Cartesian Products Using Cross Joins

Cartesian product is a mathematical term. It refers to the set of data created by merging the rows from two or more tables together. Cross join is the syntax used to create a Cartesian product by joining multiple tables. Both terms are often used synonymously.
The format of the syntax for the CROSS JOIN clause is as follows:

SELECT *table1*.column, *table2*.column
FROM *table1*
CROSS JOIN *table2*;

It is important to observe that no *join condition* is specified using the ON or USING keywords. A Cartesian product freely associates the rows from *table1* with every row in *table2.* Conditions that limit the results are permitted in the form of WHERE clause restrictions. If *table1* and *table2* contain *x* and *y* number of rows, respectively, the Cartesian product will contain *x* times *y* number of rows. The results from a *cross join* may be used to identify orphan rows or generate a large data set for use in application testing.
Consider the following queries:

```
Query 1: select * from jobs cross join job_history;
Query 2: select * from jobs j cross join job_history jh where j.job_id='AD_PRES';
```

Query 1 takes the 19 rows and 4 columns from the JOBS table and the 10 rows and 5 columns from the JOB_HISTORY table and generates one large set of 190 records with 9 columns. SQL*Plus presents any identically named columns as headings. SQL Developer appends an underscore and number to each shared column name and uses it as the heading. The JOB_ID column is common to both the JOBS and JOB_HISTORY tables. The headings in SQL Developer are labeled JOB_ID and JOB_ID_1, respectively.
Query 2 generates the same Cartesian product as the first, but the 190 rows are constrained by the WHERE clause condition and only 10 rows are returned.

When using the **cross join** ***syntax, a Cartesian product is intentionally generated. Inadvertent Cartesian products are created when there are insufficient join conditions in a statement. Joins that specify fewer than N-1*** **join conditions** ***when joining N tables or that specify invalid*** **join conditions** ***may inadvertently create Cartesian products. A*** **pure natural join** ***between two tables sharing no identically named columns results in a Cartesian join since two tables are joined but less than one condition is available.***

Performing a Cross-Join

You are required to obtain the number of rows in the EMPLOYEES and DEPARTMENTS table as well as the number of records that would be created by a Cartesian product of these two tables. Confirm your results by explicitly counting and multiplying the number of rows present in each of these tables.

```
SELECT COUNT(*).
FROM EMPLOYEES CROSS JOIN DEPARTMENTS

SELECT COUNT(*) FROM EMPLOYEES;
SELECT COUNT(*) FROM DEPARTMENTS;
```

TWO-MINUTE DRILL

Write SELECT Statements to Access Data from More Than One Table Using Equijoins and Nonequijoins

❑ *Equijoining* occurs when one query fetches column values from multiple tables in which the rows fulfill an *equality-based join condition*.

❑ A *pure natural join* is performed using the *NATURAL JOIN* syntax when the source and target tables are implicitly equijoined using all identically named columns.

❑ The *JOIN…USING* syntax allows a *natural join* to be formed on specific columns with shared names.

❑ Dot notation refers to qualifying a column by prefixing it with its table name and a dot or period symbol. This designates the table a column originates from and differentiates it from identically named columns from other tables.

❑ The *JOIN…ON* clause allows the explicit specification of *join columns* regardless of their column names. This provides a flexible joining format.

❑ The *ON, USING,* and *NATURAL* keywords are mutually exclusive and therefore cannot appear together in a join clause.

❑ A *nonequijoin* is performed when the values in the *join columns* fulfill the *join condition* based on an inequality expression.

Join a Table to Itself Using a Self-Join

❑ A *self-join* is required when the join columns originate from the same table. Conceptually, the source table is duplicated and a target table is created. The *self-join* then works as a regular join between two discrete tables.

❑ Storing hierarchical data in a relational table requires a minimum of two columns per row. One column stores an identifier of the row's parent record and the second stores the row's identifier.

View Data That Does Not Meet a Join Condition Using Outer Joins

❑ When equijoins and nonequijoins are performed, rows from the source and target tables are matched. These are referred to as *inner joins.*

❑ An *outer join* is performed when rows, which are not retrieved by an *inner join,* are included for retrieval.

❑ A *left outer join* between the source and target tables returns the results of an inner join and the missing rows it excluded from the *source* table.

❑ A *right outer join* between the source and target tables returns the results of an inner join and the missing rows it excluded from the *target* table.

❑ A *full outer join* returns the combined results of a *left outer join* and *right outer join*.

Generate a Cartesian Product of Two or More Tables

❑ A Cartesian product is sometimes called a cross join. It is a mathematical term that refers to the set of data created by merging the rows from two or more tables.

❑ The count of the rows returned from a Cartesian product is equal to the number of rows in the source table multiplied by the number of rows in the target table.

❑ Joins that specify fewer than N-1 join conditions when joining N tables, or that specify invalid join conditions, inadvertently create Cartesian products.

SELF TEST

Choose all the correct answers for each question.

Write SELECT Statements to Access Data from More Than One Table Using Equijoins and Nonequijoins

1. The EMPLOYEES and DEPARTMENTS tables have two identically named columns: DEPARTMENT_ID and MANAGER_ID. Which of these statements joins these tables based only on common DEPARTMENT_ID values? (Choose all that apply.)
A. SELECT * FROM EMPLOYEES NATURAL JOIN DEPARTMENTS;
B. SELECT * FROM EMPLOYEES E NATURAL JOIN DEPARTMENTS D ON
E.DEPARTMENT_ID=D.DEPARTMENT_ID;
C. SELECT * FROM EMPLOYEES NATURAL JOIN DEPARTMENTS USING (DEPARTMENT_ID);
D. None of the above

2. The EMPLOYEES and DEPARTMENTS tables have two identically named columns: DEPARTMENT_ID and MANAGER_ID. Which statements join these tables based on both column values? (Choose all that apply.)
A. SELECT * FROM EMPLOYEES NATURAL JOIN DEPARTMENTS;
B. SELECT * FROM EMPLOYEES JOIN DEPARTMENTS USING (DEPARTMENT_ID,MANAGER_ID);
C. SELECT * FROM EMPLOYEES E JOIN DEPARTMENTS D ON E.DEPARTMENT_
ID=D.DEPARTMENT_ID AND E.MANAGER_ID=D.MANAGER_ID;
D. None of the above

3. Which join is performed by the following query?
SELECT E.JOB_ID,J.JOB_ID FROM EMPLOYEES E JOIN JOBS J ON (E.SALARY < J.MAX_SALARY); (Choose the best answer.)
A. Equijoin
B. Nonequijoin
C. Cross join
D. Outer join

4. Which of the following statements are syntactically correct? (Choose all that apply.)
A. SELECT * FROM EMPLOYEES E JOIN DEPARTMENTS D USING (DEPARTMENT_ID);
B. SELECT * FROM EMPLOYEES JOIN DEPARTMENTS D USING (D.DEPARTMENT_ID);
C. SELECT D.DEPARTMENT_ID FROM EMPLOYEES JOIN DEPARTMENTS D USING (DEPARTMENT_ID);
D. None of the above

5. Which of the following statements are syntactically correct? (Choose all that apply.)
A. SELECT E.EMPLOYEE_ID, J.JOB_ID PREVIOUS_JOB, E.JOB_ID CURRENT_JOB FROM
JOB_HISTORY J CROSS JOIN EMPLOYEES E ON (J.START_DATE=E.HIRE_DATE);
B. SELECT E.EMPLOYEE_ID, J.JOB_ID PREVIOUS_JOB, E.JOB_ID CURRENT_JOB
FROM JOB_HISTORY J JOIN EMPLOYEES E ON (J.START_DATE=E.HIRE_DATE);
C. SELECT E.EMPLOYEE_ID, J.JOB_ID PREVIOUS_JOB, E.JOB_ID CURRENT_JOB
FROM JOB_HISTORY J OUTER JOIN EMPLOYEES E ON (J.START_DATE=E.
HIRE_DATE);
D. None of the above

6. Choose one correct statement regarding the following query:
SELECT * FROM EMPLOYEES E JOIN DEPARTMENTS D ON (D.DEPARTMENT_ID=E.DEPARTMENT_ID) JOIN
LOCATIONS L ON (L.LOCATION_ID =D.LOCATION_ID);
A. Joining three tables is not permitted.
B. A Cartesian product is generated.
C. The JOIN…ON clause may be used for joins between multiple tables.
D. None of the above

Join a Table to Itself Using a Self-Join

7. How many rows are returned after executing the following statement?
SELECT * FROM REGIONS R1 JOIN REGIONS R2 ON (R1.REGION_ID=LENGTH(R2.REGION_NAME)/2);

The REGIONS table contains the following row data. (Choose the best answer.)

1 Europe
2 Americas
3 Asia
4 Middle East and Africa
A. 2
B. 3
C. 4
D. None of the above

View Data That Does Not Meet a Join Condition Using Outer Joins

8. Choose one correct statement regarding the following query.
SELECT C.COUNTRY_ID FROM LOCATIONS L RIGHT OUTER JOIN COUNTRIES C
ON (L.COUNTRY_ID=C.COUNTRY_ID) WHERE L.COUNTRY_ID is NULL
A. No rows in the LOCATIONS table have the COUNTRY_ID values returned.
B. No rows in the COUNTRIES table have the COUNTRY_ID values returned.
C. The rows returned represent the COUNTRY_ID values for all the rows in the LOCATIONS table.
D. None of the above

9. Which of the following statements are syntactically correct? (Choose all that apply.)
A. SELECT JH.JOB_ID FROM JOB_HISTORY JH RIGHT OUTER JOIN JOBS J ON JH.JOB_ID=J.JOB_ID
B. SELECT JOB_ID FROM JOB_HISTORY JH RIGHT OUTER JOIN JOBS J ON (JH.JOB_ID=J.JOB_ID)
C. SELECT JOB_HISTORY.JOB_ID FROM JOB_HISTORY OUTER JOIN JOBS ON JOB_HISTORY.JOB_ID=JOBS.JOB_ID
D. None of the above

Generate a Cartesian Product of Two or More Tables

10. If the REGIONS table, which contains 4 rows, is cross joined to the COUNTRIES table, which contains 25 rows, how many rows appear in the final results set? (Choose the best answer.)
A. 100 rows
B. 4 rows
C. 25 rows
D. None of the above

LAB QUESTION

Using SQL Developer or SQL*Plus, connect to the OE schema and complete the following tasks. You are required to produce a report of customers who purchased products with list prices of more than $1000. The report must contain customer first and last names and the product names and their list prices. Customer information is stored in the CUSTOMERS table, which has the CUSTOMER_ID column as its primary key. The product name and list price details are stored in the PRODUCT_INFORMATION table with the PRODUCT_ID column as its primary key. Two other related tables may assist in generating the required report: the ORDERS table, which stores the CUSTOMER_ID and ORDER_ID information, and the ORDER_ITEMS table, which stores the PRODUCT_ID values associated with each ORDER_ID.
There are several approaches to solving this question. Your approach may differ from the solution listed.

SELF TEST ANSWERS

Write SELECT Statements to Access Data from More Than One Table Using Equijoins and Nonequijoins

1. ✹✓ D. The queries in **B** and **C** incorrectly contain the NATURAL keyword. If this is removed, they will join the DEPARTMENTS and EMPLOYEES tables based on the DEPARTMENT_ID column.
✹✗ **A, B,** and **C** are incorrect. **A** performs a pure natural join that implicitly joins the two tables on all columns with identical names which, in this case, are DEPARTMENT_ID and MANAGER_ID.

2. ✹✓ A, B, and **C.** These clauses demonstrate different techniques to join the tables on both the DEPARTMENT_ID and MANAGER_ID columns.
✹✗ **D** is incorrect.

3. ✹✓ B. The join condition is an expression based on the *less than* inequality operator. Therefore, this join is a nonequijoin.
✹✗ **A, C,** and **D** are incorrect. **A** would be correct if the operator in the join condition expression was an equality operator. The CROSS JOIN keywords or the absence of a join condition would result in **C** being true. **D** would be true if one of the OUTER JOIN clause was used instead of the JOIN…ON clause.

4. ✹✓ A. This statement demonstrates the correct usage of the JOIN…USING clause.
✹✗ **B, C,** and **D** are incorrect. **B** is incorrect since only nonqualified column names are allowed in the brackets after the USING keyword. **C** is incorrect because the column in brackets after the USING keyword cannot be referenced with a qualifier in the SELECT clause.

5. ✹✓ B demonstrates the correct usage of the JOIN…ON clause.
✹✗ **A, C,** and **D** are incorrect. **A** is incorrect since the CROSS JOIN clause cannot contain the ON keyword. **C** is incorrect since the OUTER JOIN keywords must be preceded by the LEFT, RIGHT, or FULL keyword.

6. ✹✓ C. The JOIN…ON clause and the other join clauses may all be used for joins between multiple tables. The JOIN…ON and JOIN…USING clauses are better suited for N-way table joins.
✹✗ **A, B,** and **D** are incorrect. **A** is false since you may join as many tables as you wish. A Cartesian product is not created since there are two join conditions and three tables.

Join a Table to Itself Using a Self-Join

7. ✹✓ B. Three rows are returned. For the row with a REGION_ID value of 2, the REGION_NAME is Asia and half the length of the REGION_NAME is also 2. Therefore this row is returned. The same logic results in the rows with REGION_ID values of three and four and REGION_NAME values of Europe and Americas being returned.
✹✗ **A, C,** and **D** are incorrect.

View Data That Does Not Meet a Join Condition Using Outer Joins

8. ✹✓ A. The right outer join fetches the COUNTRIES rows that the inner join between the LOCATIONS and COUNTRIES tables have excluded. The WHERE clause then restricts the results by eliminating the inner join results. This leaves the rows from the COUNTRIES table with which no records from the LOCATIONS table records are associated.
✹✗ **B, C,** and **D** are incorrect.

9. ✹✓ A. This statement demonstrates the correct use of the RIGHT OUTER JOIN…ON clause.
✹✗ **B, C,** and **D** are incorrect. The JOB_ID column in the SELECT clause in **B** is not qualified and is therefore ambiguous since the table from which this column comes is not specified. **C** uses an OUTER JOIN without the keywords LEFT, RIGHT, or FULL.

Generate a Cartesian Product of Two or More Tables

10. ✹✓ **A.** The cross join associates every four rows from the REGIONS table 25 times with the rows from the COUNTRIES table yielding a result set that contains 100 rows.
✹✓ **B, C,** and **D** are incorrect.

LAB ANSWER

```
SELECT CUST_FIRST_NAME, CUST_LAST_NAME, PRODUCT_NAME, LIST_PRICE.
FROM CUSTOMERS JOIN ORDERS USING (CUSTOMER_ID) JOIN ORDER_ITEMS USING (ORDER_ID)
JOIN PRODUCT_INFORMATION USING (PRODUCT_ID)WHERE LIST_PRICE > 1000.
```

Chapter 7: Using Subqueries to Solve Problems

Till now in every case the SELECT statement has been a single, self-contained command. This chapter is the first of two that show how two or more SELECT commands can be combined into one statement.
The first technique is the use of *subqueries*. A subquery is a SELECT statement whose output is used as input to another SELECT statement (or indeed to a DML statement).
The second technique is the use of set operators, where the results of several SELECT commands are combined into a single result set.

Define Subqueries

A subquery is a query that is nested inside a SELECT, INSERT, UPDATE, or DELETE statement or inside another subquery. A subquery can return a set of rows or just one row to its parent query.
A *scalar* subquery is a query that returns exactly one value: a single row, with a single column. Scalar subqueries can be used in most places in a SQL statement where you could use an expression or a literal value.
The places in a query where a subquery may be used are as follows:

- In the SELECT list used for column projection
- In the FROM clause
- In the WHERE clause
- In the HAVING clause

A subquery is often referred to as an *inner* query, and the statement within which it occurs is then called the *outer* query. There is nothing wrong with this terminology, except that it may imply that you can only have two levels, inner and outer. In fact, the Oracle implementation of subqueries does not impose any practical limits on the level of nesting: the depth of nesting permitted in the FROM clause of a statement is unlimited, and that in the WHERE clause is up to 255.
A subquery can have any of the usual clauses for selection and projection. The following are required clauses:

- A SELECT list
- A FROM clause

The following are optional clauses:

- WHERE
- GROUP BY
- HAVING

The subquery (or subqueries) within a statement must be executed before the parent query that calls it, in order that the results of the subquery can be passed to the parent.

Types of Subqueries

In this exercise, you will write code that demonstrates the places where subqueries can be used.
The query will report on the current numbers of departments and staff:

```
select sysdate Today,
(select count(*) from departments) Dept_count,
(select count(*) from employees) Emp_count from dual;
```

Write a query to identify all the employees who are managers. This will require using a subquery in the WHERE clause to select all the employees whose EMPLOYEE_ID appears as a MANAGER_ID:

```
select last_name from employees where
(employee_id in (select manager_id from employees));
```

Write a query to identify the highest salary paid in each country. This will require using a subquery in the FROM clause:

```
select max(salary),country_id from
(select salary,department_id,location_id,country_id from
employees natural join departments natural join locations)group by country_id;
```

Describe the Types of Problems That the Subqueries Can Solve

There are many situations where you will need the result of one query as the input for another.

Use of a Subquery Result Set for Comparison Purposes

Which employees have a salary that is less than the average salary? This could be answered by two statements, or by a single statement with a subquery. The following example uses two statements:

```
select avg(salary) from employees;
select last_name from employees where salary < result_of_previous_query ;
```

Alternatively, this example uses one statement with a subquery:

```
select last_name from employees
where salary < (select avg(salary)from employees);
```

In this example, the subquery is used to substitute a value into the WHERE clause of the parent query: it is returning a single value, used for comparison with the rows retrieved by the parent query. The subquery could return a set of rows. For example, you could use the following to find all departments that do actually have one or more employees assigned to them:

```
select department_name from departments where department_id in
(select distinct(department_id) from employees);
```

In the preceding example, the subquery is used as an alternative to a join. The same result could have been achieved with the following:

```
select department_name from departments inner join employees
on employees.department_id = departments.department_id group by department_name;
```

If the subquery is going to return more than one row, then the comparison operator must be able to accept multiple values. These operators are IN, NOT IN, ANY, and ALL. If the comparison operator is EQUAL, GREATER THAN, or LESS THAN (which each can only accept one value), the parent query will fail.

Using NOT IN is fraught with problems because of the way SQL handles NULLs. As a general rule, do not use NOT IN unless you are certain that the result set will not include a NULL.

Star Transformation

An extension of the use of subqueries as an alternative to a join is to enable the *star transformation* often needed in data warehouse applications. Consider a large table recording sales. Each sale is marked as being of a particular product to a particular buyer through a particular channel. These attributes are identified by codes, used as foreign keys to dimension tables with rows that describe each product, buyer, and channel. To identify all sales of books to buyers in Germany through Internet orders, one could run a query like this:

```
select … from sales s, products p, buyers b, channels c
where s.prod_code=p.prod_code
and s.buy_code=b.buy_code
and s.chan_code=c.chan_code
and p.product=’Books’
and b.country=’Germany’
and c.channel=’Internet’;
```

This query uses the WHERE clause to join the tables and then to filter the results.

The following is an alternative query that will yield the same result:

```
select … from sales
where prod_code in (select prod_code from products where product=’Books’)
and buy_code in (select buy_code from buyers where country=’Germany’)
and chan_code in (select chan_code from channels where channel=’Internet);
```

The rewrite of the first statement to the second is the star transformation. Apart from being an inherently more elegant structure (most SQL developers with any sense of aesthetics will agree with that), there are technical reasons why the database may be able to execute it more efficiently than the original query. Also, star queries are easier to maintain; it is very simple to add more dimensions to the query or to replace the single literals (‘Books,’ ‘Germany,’ and ‘Internet’) with lists of values.

There is an instance initialization parameter, STAR_TRANSFORMATION_ENABLED, which (if set to true) will permit the Oracle query optimizer to re-write code into star queries.

Generate a Table from Which to SELECT

Subqueries can also be used in the FROM clause, where they are sometimes referred to as *inline views*. Consider another problem based on the HR schema: employees are assigned to a department, and departments have a location. Each location is in a country. How can you find the average salary of staff in a country, even though they work for different departments? Like this:

```
select avg(salary),country_id from
(select salary,department_id,location_id,country_id from
employees natural join departments natural join locations)
group by country_id;
```

The subquery constructs a table with every employee’s salary and the country in which his department is based. The parent query then addresses this table, averaging the SALARY and grouping by COUNTRY_ID.

Generate Values for Projection

The third place a subquery can go is in the SELECT list of a query. How can you identify the highest salary and the highest commission rate and thus what the maximum commission paid would be if the highest salaried employee also had the highest commission rate? Like this, with two subqueries:

```
select(select max(salary) from employees) *(select max(commission_pct) from employees)
/ 100
from dual;
```

In this usage, the SELECT list used to project columns is being populated with the results of the subqueries. A subquery used in this manner must be scalar, or the parent query will fail with an error.

Generate Rows to be Passed to a DML Statement

DML statements are covered in detail in Chapter 10. For now, consider these examples:

```
insert into sales_hist select * from sales where date > sysdate-1;
update employees set salary = (select avg(salary) from employees);
delete from departments
where department_id not in (select department_id from employees);
```

The first example uses a subquery to identify a set of rows in one table that will be inserted into another. The second example uses a subquery to calculate the average salary of all employees and passes this value (a scalar quantity) to an update statement. The third example uses a subquery to retrieve all DEPARTMENT_IDs that are in use and passes the list to a DELETE command, which will remove all departments that are not in use.

Note that it is not legal to use a subquery in the VALUES clause of an insert statement; this is fine:

```
insert into dates select sysdate from dual;
```

But this is not:

```
insert into dates (date_col) values (select sysdate fom dual);
```

More Complex Subqueries

```
select last_name from employees where department_id in
(select department_id from departments where location_id in
(select location_id from locations where country_id =
(select country_id from countries where country_name=’United Kingdom’)));
```

Write a query to identify all the employees who earn more than the average and who work in any of the IT departments. This will require two subqueries, not nested:

```
select last_name from employees where department_id in
(select department_id from departments where department_name like ’IT%’)
and salary > (select avg(salary) from employees);
```

List the Types of Subqueries

There are three broad divisions of subquery:

- Single-row subqueries
- Multiple-row subqueries
- Correlated subqueries

Single- and Multiple-Row Subqueries

The *single-row* subquery returns one row. A special case is the scalar subquery, which returns a single row with one column. Scalar subqueries are acceptable (and often very useful) in virtually any situation where you could use a literal value, a constant, or an expression.

Multiple-row subqueries return sets of rows. These queries are commonly used to generate result sets that will be passed to a DML or SELECT statement for further processing. Both single-row and multiple-row subqueries will be evaluated once, before the parent query is run.

Single- and multiple-row subqueries can be used in the WHERE and HAVING clauses of the parent query, but there are restrictions on the legal comparison operators. If the comparison operator is any of the ones in the following table, the subquery must be a single-row subquery:

=	equal
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal
<>	not equal
!=	not equal

If any of the operators in the preceding table are used with a subquery that returns more than one row, the query will fail. The operators in the following table can use multiple-row subqueries:

IN	equal to any member in a list
NOT IN	not equal to any member in a list
ANY	returns rows that match any value on a list
ALL	returns rows that match all the values in a list

Correlated Subqueries

A *correlated subquery* has a more complex method of execution than single- and multiple-row subqueries and is potentially much more powerful. If a subquery references columns in the parent query, then its result will be dependent on the parent query. This makes it impossible to evaluate the subquery before evaluating the parent query.

Consider this statement, which lists all employees who earn less than the average salary:

```
select last_name from employees where salary < (select avg(salary) from employees);
```

The single-row subquery need only be executed once, and its result substituted into the parent query. But now consider a query that will list all employees whose salary is less than the average salary of their department. In this case, the subquery must be run for each employee to determine the average salary for her department; it is necessary to pass the employee's department code to the subquery. This can be done as follows:

```
select p.last_name, p.department_id from employees p
where p.salary < (select avg(s.salary) from employees s
where s.department_id=p.department_id);
```

In this example, the subquery references a column, `p.department_id`, from the select list of the parent query. This is the signal that, rather than evaluating the subquery once, it must be evaluated for every row in the parent query. To execute the query, Oracle will look at every row in EMPLOYEES and, as it does so, run the subquery using the DEPARTMENT_ID of the current employee row.

The flow of execution is as follows:

1. Start at the first row of the EMPLOYEES table.
2. Read the DEPARTMENT_ID and SALARY of the current row.
3. Run the subquery using the DEPARTMENT_ID from step 2.
4. Compare the result of step 3 with the SALARY from step 2, and return therow if the SALARY is less than the result.
5. Advance to the next row in the EMPLOYEES table.
6. Repeat from step 2.

A single-row or multiple-row subquery is evaluated once, before evaluating the outer query; a correlated subquery must be evaluated once for every row in the outer query. A correlated subquery can be single- or multiple-row, if the comparison operator is appropriate.

Correlated subqueries can be a very inefficient construct, due to the need for repeated execution of the subquery. Always try to find an alternative approach.

Investigate the Different Types of Subqueries

In this exercise, you will demonstrate problems that can occur with different types of subqueries. Use either SQL*Plus or SQL Developer. All the queries should be run when connected to the HR schema: it is assumed that the EMPLOYEES table has the standard sets of rows.

1. Log on to your database as user HR.
2. Write a query to determine who earns more than Mr. Tobias:

```
select last_name from employees where
salary > (select salary from employees where last_name='Tobias') order by last_name;
```

This will return 86 names, in alphabetical order.

3. Write a query to determine who earns more than Mr. Taylor:

```
select last_name from employees where
salary > (select salary from employees where last_name='Taylor') order by last_name;
```

This will fail with the error "ORA-01427: single-row subquery returns more than one row." The following illustration shows the last few lines of the output from step 2 followed by step 3 and the error, executed with SQL*Plus.

4. Determine why the query in step 2 succeeded but failed in step 3. The answer lies in the state of the data:

```
select count(last_name) from employees where last_name='Tobias';
select count(last_name) from employees where last_name='Taylor';
```

The use of the "greater than" operator in the queries for steps 2 and 3 requires a single-row subquery, but the subquery used may return any number of rows, depending on the search predicate used.

5. Fix the code in steps 2 and 3 so that the statements will succeed no matter what LAST_NAME is used. There are two possible solutions: one uses a different comparison operator that can handle a multiple-row subquery; the other uses a subquery that will always be single-row.

The first solution:

```
select last_name from employees where
salary > all (select salary from employees where last_name='Taylor')
order by last_name;
```

The second solution:

```
select last_name from employees where
salary > (select max(salary) from employees where last_name='Taylor')
order by last_name;
```

Write Single-Row and Multiple-Row Subqueries

Following are examples of single- and multiple-row subqueries. They are based on the HR demonstration schema.

How would you figure out which employees have a manager who works for a department based in the United Kingdom? This is a possible solution, using multiple-row subqueries:

```
select last_name from employees
```

```
where manager_id in
(select employee_id from employees where department_id in
(select department_id from departments where location_id in
(select location_id from locations where country_id='UK')));
```

In the preceding example, subqueries are nested three levels deep. Note that the subqueries use the IN operator because it is possible that the queries could return several rows.

How can you best design subqueries such that they will not fail with “ORA-01427: single-row subquery returns more than one row” errors?
There are two common techniques: use an aggregation so that if you do get multiple rows they will be reduced to one, or use one of the IN, ANY, or ALL operators so that it won’t matter if multiple rows are returned. But these are both hacker’s solutions;
The real answer is always to use the primary key when identifying the row to be returned, not a nonunique key. Sometimes there is a choice between using a subquery or using some other technique: the star transformation is a case in point. Which is better? It depends on the circumstances. It is not uncommon for the different techniques to cause a different execution method within the database. Depending on how the instance, the database, and the data
structures within it are configured, one may be much more efficient than another. Whenever such a choice arises, the statements should be subjected to a tuning analysis. Your DBA will be able to advise on this.

You have been asked to find the job with the highest average salary. This can be done with a single-row subquery:

```
select job_title from jobs natural join employees group by job_title
having avg(salary) =(select max(avg(salary)) from employees group by job_id);
```

The subquery returns a single value: the average salary of the department with the highest average salary. It is safe to use the equality operator for this subquery because the MAX function guarantees that only one row will be returned.
The ANY and ALL operators are supported syntax, but their function can be duplicated with other more commonly used operators combined with aggregations.
For example, these two statements, which retrieve all employees whose salary is above that of anyone in department 80, will return identical result sets:

```
select last_name from employees where salary > all
(select salary from employees where department_id=80);
select last_name from employees where salary >
(select max(salary) from employees where department_id=80);
```

The following summarizes the equivalents for ANY and ALL:

< ANY less than the highest
> ANY more than the lowest
= ANY equivalent to IN
> ALL more than the highest
< ALL less than the lowest

Write a Query That Is Reliable and User Friendly

In this exercise, develop a multi-row subquery that will prompt for user input. Use either SQL*Plus or SQL Developer. All the queries should be run when connected to the HR schema; it is assumed that the tables have the standard sets of rows.
1. Log on to your database as user HR.
2. Design a query that will prompt for a department name and list the last name of every employee in that department:

```
select last_name from employees where department_id =
(select department_id from departments
where department_name = '&Department_name');
```

3. Run the query in step 2 three times, when prompted supplying these values:
first time- Executive
second time- executive
third time- Executiv
4. Note the results from step 3. The first run succeeded because the value entered was an exact match, but the other failed. Adjust the query to make it more user friendly, so that it can handle minor variations in case or spelling:

```
select last_name from employees where department_id =
(select department_id from departments
where upper(department_name) like upper('%&Department_name%'));
```

5. Run the query in step 4 three times, using the same values as used in step 3. This time, the query will execute successfully.
6. Run the query in step 4 again, and this time enter the value Pu. The query will fail, with an “ORA-01427: single-row subquery returns more than one row” error, because the attempt to make it more user-friendly means that the subquery is no longer guaranteed to be a single-row subquery. The string Pu matches two departments.
7. Adjust the query to make it resilient against the ORA-01427 error, and adjust the output to prevent any possible confusion:

```
select last_name,department_name from employees join departments
on employees.department_id = departments.department_id
where departments.department_id in
(select department_id from departments
where upper(department_name) like upper('%&Department_name%'));
```

Use of Subqueries

Subqueries come in three general forms: single-row, multiple-row, and correlated. A special case of the single-row subquery is the scalar subquery, a subquery that returns exactly one value. This is a single-row singlecolumn subquery. For the first SQL OCP exam, detailed knowledge is expected only of scalar subqueries and single-column multiple-row subqueries. Correlated subqueries and multiple column subqueries are unlikely to be examined at this level, but a general knowledge of them may be tested.
When using subqueries in a WHERE clause, you must be aware of which operators will succeed with single-row subqueries and which will succeed with multiple-row subqueries.

✓ TWO-MINUTE DRILL

Define Subqueries

❑ A subquery is a select statement embedded within another SQL statement.

❑ Subqueries can be nested within each other.

❑ With the exception of the correlated subquery, subqueries are executed before the outer query within which they are embedded.

Describe the Types of Problems That the Subqueries Can Solve

❑ Selecting rows from a table with a condition that depends on the data within the table can be implemented with a subquery.

❑ Complex joins can sometimes be replaced with subqueries.

❑ Subqueries can add values to the outer query's output that are not available in the tables the outer query addresses.

List the Types of Subqueries

❑ Multiple-row subqueries can return several rows, possibly with several columns.

❑ Single-row subqueries return one row, possibly with several columns.

❑ A scalar subquery returns a single value; it is a single-row, single-column subquery.

❑ A correlated subquery is executed once for every row in the outer query.

Write Single-Row and Multiple-Row Subqueries

❑ Single-row subqueries should be used with single-row comparison operators.

❑ Multiple-row subqueries should be used with multiple-row comparison operators.

❑ The ALL and ANY operators can be alternatives to use of aggregations.

SELF TEST

Define Subqueries

1. Consider this generic description of a SELECT statement:
SELECT *select_list*
FROM *table*
WHERE *condition*
GROUP BY *expression_1*
HAVING *expression_2*
ORDER BY *expression_3* ;
Where could subqueries be used? (Choose all correct answers.)
A. *select_list*
B. *table*
C. *condition*
D. *expression_1*
E. *expression_2*
F. *expression_3*

2. A query can have a subquery embedded within it. Under what circumstances could there be more than one subquery? (Choose the best answer.)
A. The outer query can include an inner query. It is not possible to have another query within the inner query.
B. It is possible to embed a single-row subquery inside a multiple-row subquery, but not the other way around.
C. The outer query can have multiple inner queries, but they must not be embedded within each other.
D. Subqueries can be embedded within each other with no practical limitations on depth.

3. Consider this statement:

```
select employee_id, last_name from employees where
salary > (select avg(salary) from employees);
```

When will the subquery be executed? (Choose the best answer.)
A. It will be executed before the outer query.
B. It will be executed after the outer query.
C. It will be executed concurrently with the outer query.
D. It will be executed once for every row in the EMPLOYEES table.

4. Consider this statement:

```
select o.employee_id, o.last_name from employees o where
o.salary > (select avg(i.salary) from employees i
where i.department_id=o.department_id);
```

When will the subquery be executed? (Choose the best answer.)
A. It will be executed before the outer query.
B. It will be executed after the outer query.
C. It will be executed concurrently with the outer query.
D. It will be executed once for every row in the EMPLOYEES table.

Describe the Types of Problems That the Subqueries Can Solve

5. Consider the following statement:

```
select last_name from employees join departments
on employees.department_id = departments.department_id
where department_name='Executive';
```

and this statement:

```
select last_name from employees where department_id in
(select department_id from departments where department_name='Executive');
```

What can be said about the two statements? (Choose two correct answers.)
A. The two statements should generate the same result.
B. The two statements could generate different results.
C. The first statement will always run successfully; the second statement will error if there are two departments with DEPARTMENT_NAME 'Executive.'
D. Both statements will always run successfully, even if there are two departments with DEPARTMENT_NAME 'Executive.'

List the Types of Subqueries

6. What are the distinguishing characteristics of a scalar subquery? (Choose two correct answers.)
A. A scalar subquery returns one row.
B. A scalar subquery returns one column.
C. A scalar subquery cannot be used in the SELECT LIST of the parent query.
D. A scalar subquery cannot be used as a correlated subquery.

7. Which comparison operator cannot be used with multiple-row subqueries? (Choose the best answer.)
A. ALL
B. ANY
C. IN
D. NOT IN
E. All the above can be used.

Write Single-Row and Multiple-Row Subqueries

8. Consider this statement:

```
select last_name, (select count(*) from departments) from employees
where salary = (select salary from employees);
```

What is wrong with it? (Choose the best answer.)

A. Nothing is wrong—the statement should run without error.

B. The statement will fail because the subquery in the SELECT list references a table that is not listed in the FROM clause.

C. The statement will fail if the second query returns more than one row.

D. The statement will run but is extremely inefficient because of the need to run the second subquery once for every row in EMPLOYEES.

9. Which of the following statements are equivalent? (Choose two answers.)

A. select employee_id from employees where salary < all (select salary from employees where department_id=10);

B. select employee_id from employees where salary < (select min(salary) from employees where department_id=10);

C. select employee_id from employees where salary not >= any (select salary from employees where department_id=10);

D. select employee_id from employees e join departments d on e.department_id=d.department_id where e.salary < (select min(salary) from employees) and d.department_id=10;

10. Consider this statement, which is intended to prompt for an employee’s name and then find all employees who have the same job as the first employee:

```
select last_name,employee_id from employees where job_id =
(select job_id from employees where last_name = ’&Name’);
```

What would happen if a value were given for &Name that did not match with any row in EMPLOYEES? (Choose the best answer.)

A. The statement would fail with an error.

B. The statement would return every row in the table.

C. The statement would return no rows.

D. The statement would return all rows where JOB_ID is NULL.

LAB QUESTION

Exercise 8-3 included this query that attempted to find all employees whose salary is higher than that of a nominated employee:

```
select last_name from employees where
salary > (select salary from employees where last_name=’Taylor’)
order by last_name;
```

The query runs successfully if `last_name` is unique. Two variations were given that will run without error no matter what value is provided.

The first solution was as follows:

```
select last_name from employees where
salary > all (select salary from employees where last_name=’Taylor’)
order by last_name;
```

The second solution was as follows:

```
select last_name from employees where
salary > (select max(salary) from employees where last_name=’Taylor’)
order by last_name;
```

There are other queries that will run successfully; construct two other solutions, one using the ANY comparison operator, the other using the MIN aggregation function. Now that you have four solutions, do they all give the same result?

All these “solutions” are in fact just ways of avoiding error. They do not necessarily give the result the user wants, and they may not be consistent. What change needs to be made to give a consistent, unambiguous, result?

SELF TEST ANSWERS

Define Subqueries

1. ✹✓ **A, B, C, D, E.** Subqueries can be used at all these points.

✹⋎ **F.** A subquery cannot be used in the ORDER BY clause of a query.

2. ✹✓ **D.** Subquery nesting can be done to many levels.

✹⋎ **A, B,** and **C. A** and **C** are wrong because subqueries can be nested. **B** is wrong because the number of rows returned is not relevant to nesting subqueries, only to the operators being used.

3. ✹✓ **A.** The result set of the inner query is needed before the outer query can run.

✹⋎ **B, C,** and **D. B** and **C** are not possible because the result of the subquery is needed before the parent query can start. **D** is wrong because the subquery is only run once.

4. ✹✓ **D.** This is a correlated subquery, which must be run for every row in the table.

✹⋎ **A, B,** and **C.** The result of the inner query is dependent on a value from the outer query; it must therefore be run once for every row.

Describe the Types of Problems That the Subqueries Can Solve

5. ✹✓ **A, D.** The two statements will deliver the same result, and neither will fail if the name is duplicated.

✹⋎ **B, C. B** is wrong because the statements are functionally identical, though syntactically different. **C** is wrong because the comparison operator used, IN, can handle a multiple-row subquery.

List the Types of Subqueries

6. ✹✓ **A, B.** A scalar subquery can be defined as a query that returns a single value.

✹⋎ **C, D. C** is wrong because a scalar subquery is the only subquery that can be used in the SELECT LIST. **D** is wrong because scalar subqueries can be correlated.

7. ✹✓ **E.** ALL, ANY, IN, and NOT IN are the multiple-row comparison operators.

✹⋎ **A, B, C, D.** All of these can be used.

Write Single-Row and Multiple-Row Subqueries

8. ✹✓ **C.** The equality operator requires a single-row subquery, and the second subquery could return several rows.

✹⋎ **A, B, D. A** is wrong because the statement will fail in all circumstances except the unlikely case where there is zero or one employees. **B** is wrong because this is not a problem; there need be no relationship between the source of data for the inner and outer queries. **D** is wrong because the subquery will only run once; it is not a correlated subquery.

9. ✹✓ **A** and **B** are identical.

✹⋎ **C** is logically the same as **A** and **B** but syntactically is not possible; it will give an error. **D** will always return no rows, because it asks for all employees who have a salary lower than all employees. This is not an error but can never return any rows. The filter on DEPARTMENTS is not relevant.

10. ✹✓ **C.** If a subquery returns NULL, then the comparison will also return NULL, meaning that no rows will be retrieved.

✹✓ **A, B, D. A** is wrong because this would not cause an error. **B** is wrong because a comparison with NULL will return nothing, not everything. **D** is wrong because a comparison with NULL can never return anything, not even other NULLs.

LAB ANSWER

The following are two possible solutions using ANY and MIN:

```
select last_name from employees where
salary > any (select salary from employees where last_name=’Taylor’)
order by last_name;
select last_name from employees where
salary not < (select min(salary) from employees where last_name=’Taylor’)
order by last_name;
```

These are just as valid as the solutions presented earlier that used ALL and MAX, but they do not give the same result. There is no way to say that these are better or worse than the earlier solutions.

The problem is that the subquery is based on a column that is not the primary key. It would not be unreasonable to say that all these solutions are wrong, and the original query is the best; it gives a result that is unambiguously correct if the LAST_NAME is unique, and if LAST_NAME is not unique, it throws an error rather than giving a questionable answer. The real answer is that the query should be based on EMPLOYEE_ID, not LAST_NAME.

All SELECT statements return a set of rows. The set operators take as their input the results of two or more SELECT statements and from these generate a single result set. This is known as a *compound query*.
Oracle provides three set operators: UNION, INTERSECT, and MINUS.
UNION can be qualified with ALL. There is a significant deviation from the ISO standard for SQL here, in that ISO SQL uses EXCEPT where Oracle uses MINUS, but the functionality is identical. The difference can be important when porting applications (or skills) developed for a third-party database to the Oracle environment.

Describe the Set Operators

The set operators used in compound queries are as follows:

- **UNION** Returns the combined rows from two queries, sorting them and removing duplicates.
- **UNION ALL** Returns the combined rows from two queries without sorting or removing duplicates.
- **INTERSECT** Returns only the rows that occur in both queries' result sets, sorting them and removing duplicates.
- **MINUS** Returns only the rows in the first result set that do not appear in the second result set, sorting them and removing duplicates.

These commands are equivalent to the standard operators used in mathematics set theory, often depicted graphically as Venn diagrams.

Set Operator General Principles

All set operators make compound queries by combining the result sets from two or more queries. If a SELECT statement includes more than one set operator (and therefore more than two queries), they will be applied in the order the programmer specifies: top to bottom and left to right. Although pending enhancements to ISO SQL will give INTERSECT a higher priority than the others, there is currently no priority of one operator over another. To override this precedence, based on the order in which the operators appear, you can use parentheses: operators within brackets will be evaluated before passing the results to operators outside the brackets.
Each query in a compound query will project its own list of selected columns. These lists must have the same number of elements, be nominated in the same sequence, and be of broadly similar data type. They do not have to have the same names (or column aliases), nor do they need to come from the same tables (or subqueries). If the column names (or aliases) are different, the result set of the compound query will have columns named as they were in the first query.
While the selected column lists do not have to be exactly the same data type, they must be from the same data type group.
For example, the columns selected by one query could be of data types DATE and NUMBER, and those from the second query could be TIMESTAMP and INTEGER. The result set of the compound query will have columns with the higher level of precision: in this case, they would be TIMESTAMP and NUMBER. Other than accepting data types from the same group, the set operators will not do any implicit type casting. If the second query retrieved columns of type VARCHAR2, the compound query would throw an error even if the string variables could be resolved to legitimate date and numeric values.
UNION, MINUS, and INTERSECT will always combine the results sets of the input queries, then sort the results to remove duplicate rows. The sorting is based on all the columns, from left to right. If all the columns in two rows have the same value, then only the first row is returned in the compound result set.
A side effect of this is that the output of a compound query will be sorted. If the sort order (which is ascending, based on the order in which the columns happen to appear in the select lists) is not the order you want, it is possible to put a single ORDER BY clause at the end of the compound query. It is not possible to use ORDER BY in any of the queries that make up the whole compound query, as this would disrupt the sorting that is necessary to remove duplicates.
UNION ALL is the exception to the sorting-no-duplicates rule: the result sets of the two input queries will be concatenated to form the result of the compound query. But you still can't use ORDER BY in the individual queries; it can only appear at the end of the compound query where it will be applied to the complete result set.

Describe the Set Operators

In this exercise, you will see the effect of the set operators.
1. Connect to your database as user HR.
2. Run this query:

```
select region_name from regions;
```

Note the result, in particular the order of the rows. If the table is as originally created, there will be four rows returned. The order will be Europe, America, Asia, Middle East.
3. Query the Regions table twice, using UNION:

```
select region_name from regions union select region_name from regions;
```

The rows returned will be as for step 1 but sorted alphabetically.
4. This time, use UNION ALL:

```
select region_name from regions union all select region_name from regions;
```

There will be double the number of rows, and they will not be sorted.
5. An intersection will retrieve rows common to two queries:

```
select region_name from regions intersect select region_name from regions;
```

All four rows are common, and the result is sorted.
6. A MINUS will remove common rows:

```
select region_name from regions minus select region_name from regions;
```

The second query will remove all the rows in the first query. Result: no rows left.

Use a Set Operator to Combine Multiple Queries into a Single Query

Compound queries are two or more queries, linked with one or more set operators. The end result is a single result set.

The examples that follow are based on two tables, OLD_DEPT and NEW_DEPT. The table OLD_DEPT is intended to represent a table created with an earlier version of Oracle, when the only data type available for representing date and time data was DATE, the only option for numeric data was NUMBER, and character data was fixed-length CHAR. The table NEW_DEPT uses the more closely defined INTEGER numeric data type (which Oracle implements as a NUMBER of up to 38 significant digits but no decimal places), the more space-efficient VARCHAR2 for character data, and the TIMESTAMP data type, which can by default store date and time values with six decimals of precision on the seconds. There are two rows in each table.

The UNION ALL Operator

A UNION ALL takes two result sets and concatenates them together into a single result set. The result sets come from two queries that must select the same number of columns, and the corresponding columns of the two queries (in the order in which they are specified) must be of the same data type group. The columns do not have to have the same names.

The UNION Operator

A UNION performs a UNION ALL and then sorts the result across all the columns and removes duplicates.

The INTERSECT Operator
The intersection of two sets is the rows that are common to both sets.

The MINUS Operator
A MINUS runs both queries, sorts the results, and returns only the rows from the first result set that do not appear in the second result set.

Using the Set Operators
A compound query is one query made up of several queries, but they are not subqueries. A subquery generates a result set that is used by another query; the queries in a compound query run independently, and a separate stage of execution combines the result sets. This combining operation is accomplished by sorting the result sets and merging them together. There is an exception to this: UNION ALL does no processing after running the two queries; it simply lists the results of each.

More Complex Examples
If two queries do not return the same number of columns, it may still be possible to un them in a compound query by generating additional columns with NULL values.
For example, consider a classification system for animals: all animals have a name and a weight, but the birds have a wingspan whereas the cats have a tail length. A query to list all the birds and cats might be:

```
select name,tail_length,to_char(null) from cats
union all
select name,to_char(null),wing_span from birds;
```

Note the use of TO_CHAR(NULL) to generate the missing values.
A compound query can consist of more than two queries, in which case operator precedence can be controlled with parentheses. Without parentheses, the set operators will be applied in the sequence in which they are specified. Consider the situation where there is a table PERMSTAFF with a listing of all permanent staff members and a table CONSULTANTS with a listing of consultant staff. There is also a table BLACKLIST of people blacklisted for one reason or another.
The following query will list all the permanent and consulting staff in a certain geographical, removing those on the blacklist:

```
select name from permstaff where location = 'Germany'
union all
select name from consultants where work_area = 'Western Europe'
minus
select name from blacklist;
```

Note the use of UNION ALL, because is assumed that no one will be in both the PERMSTAFF and the CONSULTANTS tables; a UNION would force an unnecessary sort. The order of precedence for set operators is the order specified by the programmer, so the MINUS operation will compare the BLACKLIST with the result of the UNION ALL. The result will be all staff (permanent and consulting) who do not appear on the blacklist. If the blacklisting could be applied only to consulting staff and not to permanent staff, there would be two possibilities.
First, the queries could be listed in a different order:

```
select name from consultants where work_area = 'Western Europe'
minus
select name from blacklist
union all
select name from permstaff where location = 'Germany';
```

This would return consultants who are not blacklisted and then append all permanent staff. Alternatively, parentheses could control the precedence explicitly:

```
select name from permstaff where location = 'Germany'
union all
(select name from consultants where work_area = 'Western Europe'
minus
select name from blacklist);
```

This query will list all permanent staff and then append all consultant staff who are not blacklisted.
These two queries will return the same rows, but the order will be different because the UNION ALL operations list the PERMSTAFF and CONSULTANTS tables in a different sequence. To ensure that the queries return identical result sets, there would need to be an ORDER BY clause at the foot of the compound queries.

How can you present several tables with similar data as one table?
This is a common problem, often caused by bad systems analysis or perhaps by attempts to integrate systems together. Compound queries are often the answer. By using type casting functions to force columns to the same data type and TO_CHAR(NULL) to generate missing columns, you can present the data as though it were from one table.
Are there performance issues with compound queries?
Perhaps. With the exception of UNION ALL, compound queries have to sort data, across the full width of the rows. This may be expensive in both memory and CPU. Also, if the two queries both address the same table, there will be two passes through the data as each query is run independently; if the same result could be achieved with one (though probably more complicated) query, this would usually be a faster solution. Compound queries are a powerful tool but should be used with caution.

Using the Set Operators
In this exercise, you will run more complex compound queries.
1. Connect to your database as user HR.
2. Run this query to count the employees in three departments:

```
select department_id, count(1) from employees
where department_id in (20,30,40)
group by department_id;
```

3. Obtain the same result with a compound query:

```
select 20,count(1) from employees where department_id=20
union all
select 30,count(1) from employees where department_id=30
union all
select 40,count(1) from employees where department_id=40;
```

4. Find out if any managers manage staff in both departments 20 and 30, and exclude any managers with staff in department 40:

```
select manager_id from employees where department_id=20
intersect
select manager_id from employees where department_id=30
minus
```

```
select manager_id from employees where department_id=40;
```

5. Use a compound query to report salaries subtotaled by department, by manager, and the overall total:

```
select department_id,to_number(null),sum(salary) from employees
group by department_id
union
select to_number(null),manager_id,sum(salary) from employees
group by manager_id
union all
select to_number(null),to_number(null),sum(salary) from
employees;
```

Control the Order of Rows Returned

By default, the output of a UNION ALL compound query is not sorted at all: the rows will be returned in groups in the order of which query was listed first and within the groups in the order that they happen to be stored. The output of any other set operator will be sorted in ascending order of all the columns, starting with the first column named.

It is not syntactically possible to use an ORDER BY clause in the individual queries that make up a compound query. This is because the execution of most compound queries has to sort the rows, which would conflict with the ORDER BY.

It might seem theoretically possible that a UNION ALL (which does not sort the rows) could take an ORDER BY for each query, but the Oracle implementation of UNION ALL does not permit this.

There is no problem with placing an ORDER BY clause at the end of the compound query, however. This will sort the entire output of the compound query.

The default sorting of rows is based on all the columns in the sequence they appear. A specified ORDER BY clause has no restrictions: it can be based on any columns (and functions applied to columns) in any order.

For example:

```
SQL> select deptno,trim(dname) name from old_dept
2 union
3 select dept_id,dname from new_dept
4 order by name;
DEPTNO NAME
---------- --------------------
10 Accounts
30 Admin
20 Support
```

Note that the column names in the ORDER BY clause must be the name(s) (or, in this case, the alias) of the columns in the first query of the compound query.

Control the Order of Rows Returned

In this exercise, you will tidy up the result of the final step in Exercise 2. This step produced a listing of salaries totaled by department and then by manager, but the results were not very well formatted.

1. Connect to your database as user HR.

2. Generate better column headings for the query:

```
select department_id dept,to_number(null) mgr,sum(salary)
from employees
group by department_id
union all
select to_number(null),manager_id,sum(salary) from employees
group by manager_id
union all
select to_number(null),to_number(null),sum(salary) from
employees;
```

3. Attempt to sort the results of the queries that subtotal by using UNION instead of UNION ALL:

```
select department_id dept,to_number(null) mgr,sum(salary)
from employees
group by department_id
union
select to_number(null),manager_id,sum(salary) from employees
group by manager_id
union all
select to_number(null),to_number(null),sum(salary) from
employees;
```

This would be fine, except that the subtotals for staff without a department or a manager are placed at the bottom of the output above the grand total, not within the sections for departments and managers.

4. Generate a value to replace the NULLs for department and manager codes and for the overall total:

```
select 20,count(1) from employees where department_id=20
union all
select 30,count(1) from employees where department_id=30
union all
select 40,count(1) from employees where department_id=40;
```

5. Find out if any managers manage staff in both departments 20 and 30, and exclude any managers with staff in department 40:

```
select manager_id from employees where department_id=20
intersect
select manager_id from employees where department_id=30
minus
select manager_id from employees where department_id=40;
```

6. Use a compound query to report salaries subtotaled by department, by manager, and the overall total:

```
select department_id dept,to_number(null) mgr,sum(salary)
from employees
group by department_id
union
select to_number(null),manager_id,sum(salary) from employees
group by manager_id
union all
select to_number(null),to_number(null),sum(salary) from
employees;
```

✓ TWO-MINUTE DRILL

Describe the Set Operators

❑ UNION ALL concatenates the results of two queries.

❑ UNION sorts the results of two queries and removes duplicates.

❑ INTERSECT returns only the rows common to the result of two queries.

❑ MINUS returns the rows from the first query that do not exist in the second query.

Use a Set Operator to Combine Multiple Queries into a Single Query

❑ The queries in the compound query must return the same number of columns.

❑ The corresponding columns must be of compatible data type.

❑ The set operators have equal precedence and will be applied in the order they are specified.

Control the Order of Rows Returned

❑ It is not possible to use ORDER BY in the individual queries that make a compound query.

❑ An ORDER BY clause can be appended to the end of a compound query.

❑ The rows returned by a UNION ALL will be in the order they occur in the two source queries.

❑ The rows returned by a UNION will be sorted across all their columns, left to right.

SELF TEST

Describe the Set Operators

1. Which of these set operators will not sort the rows? (Choose the best answer.)
A. INTERSECT
B. MINUS
C. UNION
D. UNION ALL

2. Which of these operators will remove duplicate rows from the final result? (Choose all that apply.)
A. INTERSECT
B. MINUS
C. UNION
D. UNION ALL

Use a Set Operator to Combine Multiple Queries into a Single Query

3. If a compound query contains both a MINUS and an INTERSECT operator, which will be applied first? (Choose the best answer.)
A. The INTERSECT, because INTERSECT has higher precedence than MINUS.
B. The MINUS, because MINUS has a higher precedence than INTERSECT.
C. The precedence is determined by the order in which they are specified.
D. It is not possible for a compound query to include both MINUS and INTERSECT.

4. There are four rows in the REGIONS table. Consider the following statements and choose how many rows will be returned for each: 0, 4, 8, or 16.
A. `select * from regions union select * from regions`
B. `select * from regions union all select * from regions`
C. `select * from regions minus select * from regions`
D. `select * from regions intersect select * from regions`

5. Consider this compound query:

```
select empno, hired from emp
union all
select emp_id,hired,fired from ex_emp;
```

The columns EMP.EMPNO and EX_EMP.EMP_ID are integer; the column EMP.HIRED is timestamp; the columns EX_EMP.HIRED and EX_EMP.FIRED are date. Why will the statement fail? (Choose the best answer.)
A. Because the columns EMPNO and EMP_ID have different names
B. Because the columns EMP.HIRED and EX_EMP.HIRED are different data types
C. Because there are two columns in the first query and three columns in the second query
D. For all the reasons above
E. The query will succeed.

Control the Order of Rows Returned

6. Which line of this statement will cause it to fail? (Choose the best answer.)
A. `select ename, hired from current_staff`
B. `order by ename`
C. minus
D. `select ename, hired from current staff`
E. where deptno=10
F. `order by ename;`

7. Study this statement:

```
select ename from emp union all select ename from ex_emp;
```

In what order will the rows be returned? (Choose the best answer.)
A. The rows from each table will be grouped and within each group will be sorted on ENAME.
B. The rows from each table will be grouped but not sorted.
C. The rows will not be grouped but will all be sorted on ENAME.
D. The rows will be neither grouped nor sorted.

LAB QUESTION

Working in the HR schema, design some queries that will generate reports using the set operators. The reports required are as follows:

1. Employees have their current job (identified by JOB_ID) recorded in their EMPLOYEES row. Jobs they have held previously (but not their current job) are recorded in JOB_HISTORY. Which employees have never changed jobs? The listing should include the employees' EMPLOYEE_ID and LAST_NAME.

2. Which employees were recruited into one job, then changed to a different job, but are now back in a job they held before? Again, you will need to construct a query that compares EMPLOYEES with JOB_HISTORY. The report should show the employees' names and the job titles. Job titles are stored in the table JOBS.

3. What jobs has any one employee held? This will be the JOB_ID for the employee's current job (in EMPLOYEES) and all previous jobs (in JOB_HISTORY). If the employee has held a job more than once, there is no need to list it more than once. Use a replacement variable to prompt for the EMPLOYEE_ID and display the job title(s). Employees 101 and 200 will be suitable employees for testing.

SELF TEST ANSWERS

Describe the Set Operators

1. ✹✓ **D.** UNION ALL returns rows in the order that they are delivered by the two queries from which the compound query is made up.

✹⮛ **A, B, C.** INTERSECT, MINUS, and UNION all use sorting as part of their execution.

2. ✹✓ **A, B, C.** INTERSECT, MINUS, and UNION all remove duplicate rows.

✹⮛ **D.** UNION ALL returns all rows, duplicates included.

Use a Set Operator to Combine Multiple Queries into a Single Query

3. ✹✓ **C.** All set operators have equal precedence, so the precedence is determined by the sequence in which they occur.

✹⮛ **A, B, D.** A and B are wrong because set operators have equal precedence—though this may change in future releases. D is wrong because many set operators can be used in one compound query.

4. ✹✓ **A** = 4; **B** = 8; **C** = 0; **D** = 4

✹⮛ Note that 16 is not used; that would be the result of a Cartesian product query.

5. ✹✓ **C.** Every query in a compound query must return the same number of columns.

✹⮛ **A, B, D, E.** A is wrong because the columns can have different names. B is wrong because the two columns are of the same data type group, which is all that was required. It therefore follows that D and E are also wrong.

Control the Order of Rows Returned

6. ✹✓ **B.** You cannot use ORDER BY for one query of a compound query; you may only place a single ORDER BY clause at the end.

✹⮛ **A, C, D, E, F.** All these lines are legal.

7. ✹✓ **B.** The rows from each query will be together, but there will be no sorting.

✹⮛ **A, C, D.** A is not possible with any syntax. C is wrong because that would be the result of a UNION, not a UNION ALL. D is wrong because UNION ALL will return the rows from each query grouped together.

LAB ANSWER

1. To identify all employees who have not changed job, query the EMPLOYEES table and remove all those who have a row in JOB_HISTORY:

```
select employee_id, last_name from employees
minus
select employee_id,last_name from
job_history join employees using (employee_id);
```

2. All employees who have changed job at least once will have a row in JOB_HISTORY; for those who are now back in a job they have held before, the JOB_ID in EMPLOYEES will be the same as the JOB_ID in one of their rows in JOB_HISTORY:

```
select last_name,job_title from employees join jobs using(job_id)
intersect
select last_name,job_title from job_history h
join jobs j on (h.job_id=j.job_id)
join employees e on (h.employee_id=e.employee_id);
```

3. This compound query will prompt for an EMPLOYEE_ID and then list the employee's current job and previous jobs,

```
select job_title from jobs join employees using (job_id)
where employee_id=&&Who
union
select job_title from jobs join job_history using (job_id)
where employee_id=&&Who;
```

Chapter 9: Manipulating Data

This is all about the commands that change data. Syntactically.

Data Manipulation Language (DML) Statement

- SELECT
- INSERT
- UPDATE
- DELETE
- MERGE

In practice, most database professionals never include SELECT as part of DML.It is considered to be a separate language. The MERGE command is often dropped as well, not because it isn't clearly a data manipulation command but because it doesn't do anything that cannot be done with other commands. MERGE can be thought of as a shortcut for executing either an INSERT or an UPDATE or a DELETE, depending on some condition.

A command often considered with DML is TRUNCATE. This is actually a DDL (Data Definition Language) command, but as the effect for end users is the same as a DELETE (though its implementation is totally different), it does fit with DML.

So, the following commands are described in the next sections, with syntax and examples:

- INSERT
- UPDATE
- DELETE
- TRUNCATE

In addition, we will discuss, for completeness:

- MERGE

INSERT

Oracle stores data in the form of rows in tables. Tables are *populated* with rows in several ways, but the most common method is with the INSERT statement. It follows that one INSERT statement can insert an individual row into one table or many rows into many tables. The basic versions of the statement do insert just one row, but more complex variations can, with one command, insert multiple rows into multiple tables.

Tables have rules defined that control the rows that may be inserted. These rules are *constraints*. A constraint is an implementation of a business rule. The business analysts who model an organization's business processes will design a set of rules for the organization's data. Examples of such rules might be that every employee must have a unique employee number, or that every employee must be assigned to a valid department.

Constraints guarantee that the data in the database conforms to the rules that define the business procedures.

There are many possible sources for the row (or rows) inserted by an INSERT statement. A single row can be inserted by providing the values for the row's columns individually. Such a statement can be constructed by typing it into SQL*Plus or SQL Developer, or by a more sophisticated user process that presents a form that prompts for values. This is the technique used for generating the rows inserted interactively by end users.

For inserts of multiple rows, the source of the rows can be a SELECT statement. The output of any and all of the SELECT statements can be used as the input to an INSERT statement. The end result of any SELECT statement can be thought of as a table: a two dimensional set of rows. This "table" can be displayed to a user or it can be passed to an INSERT command for populating another table, defined within the database. Using a SELECT statement to construct rows for an INSERT statement is a very common technique.

UPDATE

The UPDATE command is used to change rows that already exist—rows that have been created by an INSERT command, or possibly by a tool such as Datapump. As with any other SQL command, an UPDATE can affect one row or a set of rows. The size of the set affected by an UPDATE is determined by a WHERE clause, in exactly the same way that the set of rows retrieved by a SELECT statement is defined by a WHERE clause. The syntax is identical. All the rows updated will be in one table; it is not possible for a single update command to affect rows in multiple tables.

When updating a row or a set of rows, the UPDATE command specifies which columns of the row(s) to update. It is not necessary (or indeed common) to update every column of the row. If the column being updated already has a value, then this value is replaced with the new value specified by the UPDATE command. If the column was not previously populated—which is to say, its value was NULL—then it will be populated after the UPDATE with the new value.

If you select the rows to be updated with any column other than the primary key, you may update several rows, not just one. If you omit the WHERE clause completely, you will update the whole table.

An UPDATE command must honor any constraints defined for the table. For example, it will not be possible to update a column that has been marked as mandatory to a NULL value or to update a primary key column so that it will no longer be unique.

DELETE

Previously inserted rows can be removed from a table with the DELETE command. The command will remove one row or a set of rows from the table, depending on a WHERE clause. If there is no WHERE clause, every row in the table will be removed (which can be a little disconcerting if you left out the WHERE clause by mistake).

A deletion is all or nothing. It is not possible to nominate columns. When rows are inserted, you can choose which columns to populate. When rows are updated, you can choose which columns to update. But a deletion applies to the whole row—the only choice is which rows in which table. This makes the DELETE command syntactically simpler than the other DML commands.

There are no "warning" prompts for any SQL commands. If you instruct the database to delete a million rows, it will do so. Immediately. There is none of that "Are you sure?" business that some environments offer.

MERGE

It executes either an UPDATE or an INSERT or DELETE, depending on some condition. There are many occasions where you want to take a set of data (the source) and integrate it into an existing table (the target). If a row in the source data already exists in the target table, you may want to update the target row, or you may want to replace it completely, or you may want to leave the target row unchanged. If a row in the source does not exist in the target, you will want to insert it. The MERGE command lets you do this. A MERGE passes through the source data, for each row attempting to locate a matching row in the target. If no match is found, a row can be inserted; if a match is found, the matching row can be updated. The target row can even be deleted, after being matched and updated. The end result is a target table into which the data in the source has been merged.

A MERGE operation does nothing that could not be done with INSERT, UPDATE, and DELETE statements—but with one pass through the source data, it can do all three. Alternative code without a MERGE would require three passes through the data, one for each command.

The source data for a MERGE statement can be a table or any subquery. The condition used for finding matching rows in the target is similar to a WHERE clause. The clauses that update or insert rows are as complex as an UPDATE or an INSERT command.

TRUNCATE

The TRUNCATE command is not a DML command; it is DDL command. The difference is enormous. When DML commands affect data, they insert, update, and delete rows as part of transactions.

From the user's point of view, a truncation of a table is equivalent to executing a DELETE of every row: a DELETE command without a WHERE clause. But whereas a deletion may take some time (possibly hours, if there are many rows in the table) a truncation will go through instantly. It makes no difference whether the table contains one row or billions; a TRUNCATE will be virtually instantaneous. The table will still exist, but it will be empty.

DDL commands, such as TRUNCATE, will fail if there is any DML command active on the table. A transaction will block the DDL command until the DML command is terminated with a COMMIT or a ROLLBACK.

Transactions, consisting of INSERT, UPDATE, and DELETE (or even MERGE) commands can be made permanent (with a COMMIT) or reversed (with a ROLLBACK). A TRUNCATE command, like any other DDL command, is immediately permanent: it can never be reversed.

Insert Rows into a Table

The simplest form of the INSERT statement inserts one row into one table, using values provided in line as part of the command. The syntax is as follows:

INSERT INTO *table* [(*column* [,*column*…])] VALUES (*value* [,*value*…]);

For example:

```
insert into hr.regions values (10,'Great Britain');
insert into hr.regions (region_name, region_id) values ('Australasia',11);
insert into hr.regions (region_id) values (12);
insert into hr.regions values (13,null);
```

The first of the preceding commands provides values for both the columns of the REGIONS table. If the table had a third column, the statement would fail because it relies upon *positional notation*. The statement does not say which value should be inserted into which column; it relies on the position of the values: their ordering in the command. When the database receives a statement using positional notation, it will match the order of the values to the order in which the columns of the table are defined. The statement would also fail if the column order were wrong: the database would attempt the insertion but would fail because of data type mismatches.

The second command nominates the columns to be populated and the values with which to populate them. Note that the order in which columns are mentioned now becomes irrelevant—as long as the order of the columns is the same as the order of the values.

The third example lists one column, and therefore only one value. All other columns will be left null. This statement would fail if the REGION_NAME column were not nullable.

The fourth example will produce the same result, but because there is no column list, a value of some sort must be provided for each column—at the least, a NULL.

It is often considered good practice not to rely on positional notation and instead always to list the columns. This is more work but makes the code self-documenting and also makes the code more resilient against table structure changes. For instance, if a column is added to a table, all the INSERT statements that rely on positional notation will fail until they are rewritten to include a NULL for the new column. INSERT code that names the columns will continue to run.

Very often, an INSERT statement will include functions to do type casting or other editing work.

```
insert into employees (employee_id, last_name, hire_date)
values (1000,'WATSON','03-Nov-07');

insert into employees (employee_id, last_name, hire_date)
values (1000,upper('Watson'),to_date('03-Nov-07','dd-mon-yy'));
```

The rows inserted with each statement would be identical. But the first will insert exactly the literals provided. It may well be that the application relies on employee surnames being in uppercase—without this, perhaps sort orders will be wrong and searches on surname will give unpredictable results. Also, the insertion of the date value relies on automatic type casting of a string to a date, which is always bad for performance and can result in incorrect values being entered. The second statement forces the surname into uppercase whether it was entered that way or not, and specifies exactly the format mask of the date string before explicitly converting it into a date. There is no question that the second statement is a better piece of code than the first.

The following is another example of using functions:

```
insert into employees (employee_id,last_name,hire_date)
values (1000 + 1,user,sysdate - 7);
```

In the preceding statement, the EMPLOYEE_ID column is populated with the result of some arithmetic, the LAST_NAME column is populated with the result of the function USER (which returns the database logon name of the user), and the HIRE_DATE column is populated with the result of a function and arithmetic: the date seven days before the current system date.

Using functions to preprocess values before inserting rows can be particularly important when running scripts with substitution variables, as they will allow the code to correct many of the unwanted variations in data input that can occur when users enter values interactively.

To insert many rows with one INSERT command, the values for the rows must come from a query.

The syntax is as follows:

INSERT INTO *table* [(*column* [, *column*…])] *subquery*;

Note that this syntax does not use the VALUES keyword. If the column list is omitted, then the subquery must provide values for every column in the table.

To copy every row from one table to another, if the tables have the same column structure, a command such as this is all that is needed:

```
insert into regions_copy select * from regions;
```

This presupposes that the table REGIONS_COPY does exist (with or without any rows). The SELECT subquery reads every row from the source table, which is REGIONS, and the INSERT inserts them into the target table, which is
REGIONS_COPY.

There are no restrictions on the nature of the subquery. Any query returns (eventually) a two-dimensional array of rows; if the target table (which is also a two dimensional array) has columns to receive them, the insertion will work.

A daily maintenance job in a data warehouse that would assemble the data in a suitable form could be a script such as this:

```
truncate table department_salaries;
insert into department_salaries (department,staff,salaries)
select
coalesce(department_name,'Unassigned'),
count(employee_id),
sum(coalesce(salary,0))
```

```
from employees e full outer join departments d
on e.department_id = d.department_id
group by department_name
order by department_name;
```

The TRUNCATE command will empty the table, which is then repopulated from the subquery. The end users can be let loose on this table, and it should be impossible for them to misinterpret the contents—a simple natural join with no COALESCE functions, which might be all an end user would do, might be very misleading. By doing all the complex work in the INSERT statement, users can then run much simpler queries against the denormalized and aggregated data in the summary table.

Their queries will be fast, too: all the hard work has been done already.

To conclude the description of the INSERT command, it should be mentioned that it is possible to insert rows into several tables with one statement.

```
insert all
when 1=1 then
into emp_no_name (department_id,job_id,salary,commission_pct,hire_date)
values (department_id,job_id,salary,commission_pct,hire_date)
when department_id <> 80 then
into emp_non_sales (employee_id,department_id,salary,hire_date)
values (employee_id,department_id,salary,hire_date)
when department_id = 80 then
into emp_sales (employee_id,salary,commission_pct,hire_date)
values (employee_id,salary,commission_pct,hire_date)
select employee_id,department_id,job_id,salary,commission_pct,hire_date
from employees where hire_date > sysdate - 30;
```

To read this statement, start at the bottom. The subquery retrieves all employees recruited in the last 30 days. Then go to the top. The ALL keyword means the every row selected will be considered for insertion into all the tables following, not just into the first table for which the condition applies. The first condition is 1=1, which is always true, so every source row will create a row in EMP_NO_NAME. This is a copy of the EMPLOYEES table with the personal identifiers removed, a common requirement in a data warehouse. The second condition is DEPARTMENT_ID <> 80, which will generate a row in EMP_NON_SALES for every employee who is not in the sales department; there is no need for this table to have the COMMISSION_PCT column. The third condition generates a row in EMP_SALES for all the salesmen; there is no need for the DEPARTMENT_ID column, because they will all be in department 80.

Any SELECT statement, specified as a subquery, can be used as the source of rows passed to an INSERT. This enables insertion of many rows. Alternatively, using the VALUES clause will insert one row. The values can be literals or prompted for as substitution variables.

Use the INSERT Command

In this exercise, use various techniques to insert rows into a table.

1. Connect to the HR schema, with either SQL Developer or SQL*Plus.

2. Query the REGIONS table, to check what values are already in use for the REGION_ID column:

```
select * from regions;
```

This exercise assumes that values above 100 are not in use. If they are, adjust the values suggested below to avoid primary key conflicts.

3. Insert a row into the REGIONS table, providing the values in line:

```
insert into regions values (101,'Great Britain');
```

4. Insert a row into the REGIONS table, providing the values as substitution variables:

```
insert into regions values (&Region_number,'&Region_name');
```

When prompted, give the values 102 for the number, Australasia for the name. Note the use of quotes around the string.

5. Insert a row into the REGIONS table, calculating the REGION_ID to be one higher than the current high value. This will need a scalar subquery:

```
insert into regions values ((select max(region_id)+1 from regions), 'Oceania');
```

6. Confirm the insertion of the rows:

```
select * from regions;
```

7. Commit the insertions:

```
commit;
```

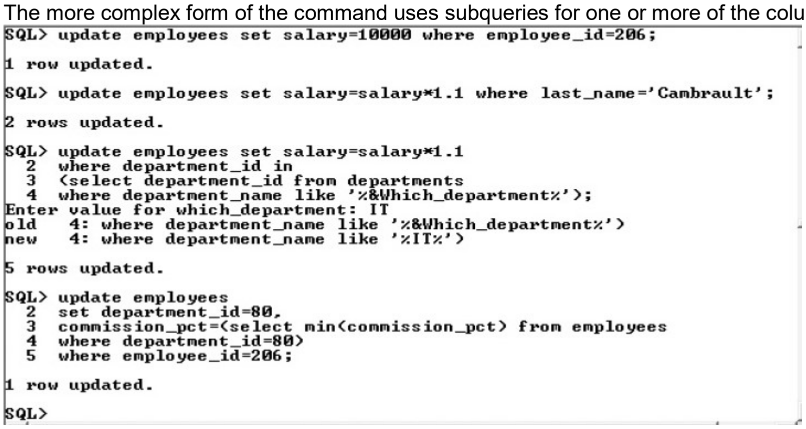
Update Rows in a Table

The UPDATE command changes column values in one or more existing rows in a single table.

The basic syntax is the following:

UPDATE *table* SET *column=value* [,*column=value*…] [WHERE *condition*];

The more complex form of the command uses subqueries for one or more of the column values and for the WHERE condition.



```
SQL> update employees set salary=10000 where employee_id=206;

1 row updated.

SQL> update employees set salary=salary*1.1 where last_name='Cambrault';

2 rows updated.

SQL> update employees set salary=salary*1.1
  2  where department_id in
  3  (select department_id from departments
  4  where department_name like '%&Which_department%');
Enter value for which_department: IT
old   4: where department_name like '%&Which_department%')
new   4: where department_name like '%IT%')

5 rows updated.

SQL> update employees
  2  set department_id=80,
  3  commission_pct=(select min(commission_pct) from employees
  4  where department_id=80)
  5  where employee_id=206;

1 row updated.

SQL>
```

The first example is the simplest. One column of one row is set to a literal value. Because the row is chosen with a WHERE clause that uses the equality predicate on the table's primary key, there is an absolute guarantee that at most only one row will be affected. No row will be changed if the WHERE clause fails to find any rows at all.

The second example shows use of arithmetic and an existing column to set the new value, and the row selection is not done on the primary key column. If the selection is not done on the primary key, or if a nonequality predicate (such as BETWEEN) is used, then the number of rows updated may be more than one. If the WHERE clause is omitted entirely, the update will be applied to every row in the table.

The third example introduces the use of a subquery to define the set of rows to be updated. A minor additional complication is the use of a replacement variable to prompt the user for a value to use in the WHERE clause of the subquery. In this example, the subquery (lines 3 and 4) will select every employee who is in a department whose name includes the string 'IT' and increment their current salary by 10 percent (unlikely to happen in practice).

It is also possible to use subqueries to determine the value to which a column will be set, as in the fourth example. In this case, one employee (identified by primary key, in line 5) is transferred to department 80 (the sales department), and then the subquery in lines 3 and 4 set his commission rate to whatever the lowest commission rate in the department happens to be.

The syntax of an update that uses subqueries is as follows:

UPDATE *table*
SET *column*=[*subquery*] [,*column*=*subquery*…]
WHERE *column* = (*subquery*) [AND *column*=*subquery*…] ;

There is a rigid restriction on the subqueries using update columns in the SET clause: the subquery must return a *scalar* value. A scalar value is a single value of whatever data type is needed: the query must return one row, with one column. If the query returns several values, the UPDATE will fail.

Consider these two examples:

```
update employees
set salary=(select salary from employees where employee_id=206);
update employees
set salary=(select salary from employees where last_name='Abel');
```

The first example, using an equality predicate on the primary key, will always succeed. Even if the subquery does not retrieve a row (as would be the case if there were no employee with EMPLOYEE_ID equal to 206), the query will still return a scalar value: a null. In that case, all the rows in EMPLOYEES would have their SALARY set to NULL—which might not be desired but is not an error as far as SQL is concerned.

The second example uses an equality predicate on the LAST_NAME, which is not guaranteed to be unique. The statement will succeed if there is only one employee with that name, but if there were more than one it would fail with the error "ORA-01427: single-row subquery returns more than one row." For code that will work reliably, no matter what the state of the data, it is vital to ensure that the subqueries used for setting column values are scalar.

A common fix for making sure that queries are scalar is to use MAX or MIN. This version of the statement will always succeed:

```
update employees
set salary=(select max(salary) from employees where last_name='Abel');
```

However, just because it will work, doesn't necessarily mean that it does what is wanted.

The subqueries in the WHERE clause must also be scalar, if it is using the equality predicate (as in the preceding examples) or the greater/less than predicates. If it is using the IN predicate, then the query can return multiple rows, as in this example which uses IN:

```
update employees
set salary=10000
where department_id in (select department_id from departments
where department_name like '%IT%');
```

This will apply the update to all employees in a department whose name includes the string 'IT.' There are several of these. But even though the query can return several rows, it must still return only one column.

Use the UPDATE Command

```
update regions set region_name='Scandinavia' where region_id=101;
```

Update a set of rows, using a nonequality predicate:

```
update regions set region_name='Iberia' where region_id > 100;
```

Update a set of rows, using subqueries to select the rows and to provide values:

```
update regions
set region_id=(region_id+(select max(region_id) from regions))
where region_id in (select region_id from regions where
region_id > 100);
```

Delete Rows from a Table

To remove rows from a table, there are two options: the DELETE command and the TRUNCATE command.

DELETE is less drastic, in that a deletion can be rolled back whereas a truncation cannot be. DELETE is also more controllable, in that it is possible to choose which rows to delete, whereas a truncation always affects the whole table. DELETE is, however, a lot slower and can place a lot of strain on the database. TRUNCATE is virtually instantaneous and effortless.

Removing Rows with DELETE

The DELETE commands removes rows from a single table. The syntax is as follows:

DELETE FROM *table* [WHERE *condition*];

```
delete from employees where employee_id=206;
delete from employees where last_name like 'S%';
delete from employees where department_id=&Which_department;
delete from employees where department_id is null;
```

The first statement identifies a row by primary key. One row only will be removed—or no row at all, if the value given does not find a match. The second statement uses a nonequality predicate that could result in the deletion of many rows: every employee whose surname begins with an uppercase "S." The third statement uses an equality predicate but not on the primary key. It prompts for a department number with a substitution variable, and all employees in that department will go. The final statement removes all employees who are not currently assigned to a department.

The condition can also be a subquery:

```
delete from employees where department_id in
(select department_id from departments where location_id in
(select location_id from locations where country_id in
(select country_id from countries where region_id in
(select region_id from regions where region_name='Europe')
)
)
)
```

This example uses a subquery for row selection that navigates the HR geographical tree (with more subqueries) to delete every employee who works for any department that is based in Europe. The same rule for the number of values returned by the subquery applies as for an UPDATE command: if the row selection is based on an equality predicate (as in the preceding example) the subquery must be scalar, but if it uses IN the subquery can return several rows.

If the DELETE command finds no rows to delete, this is not an error. The command will return the message "0 rows deleted" rather than an error message because the statement did complete successfully—it just didn't find anything to do.

Use the DELETE Command

Remove one row, using the equality predicate on the primary key:

```
delete from regions where region_id=204;
```

Attempt to remove every row in the table by omitting a WHERE clause:

```
delete from regions;
```

This will fail, due to a constraint violation.

Remove rows with the row selection based on a subquery:

```
delete from regions where
region_id in (select region_id from regions where region_name='Iberia');
```

This will return the message “2 rows deleted.”

Removing Rows with TRUNCATE

TRUNCATE is a DDL (Data Definition Language) command. It operates within the data dictionary and affects the structure of the table, not the contents of the table. However, the change it makes to the structure has the side effect of destroying all the rows in the table.

One part of the definition of a table as stored in the data dictionary is the table’s physical location. When first created, a table is allocated a single area of space, of fixed size, in the database’s data files. This is known as an *extent* and will be empty. Then, as rows are inserted, the extent fills up. Once it is full, more extents will be allocated to the table automatically.

A table therefore consists of one or more extents, which hold the rows. As well as tracking the extent allocation, the data dictionary also tracks how much of the space allocated to the table has been used. This is done with the *high water mark*. The high water mark is the last position in the last extent that has been used; all space below the high water mark has been used for rows at one time or another, and none of the space above the high water mark has been used yet.

A truncation is fast: virtually instantaneous, irrespective of whether the table has many millions of rows or none. A deletion may take seconds, minutes, hours—and it places much more strain on the database than a truncation. But a truncation is all or nothing.

The syntax to truncate a table couldn’t be simpler:

TRUNCATE TABLE *table*;

MERGE

The MERGE command is often ignored, because it does nothing that cannot be done with INSERT, UPDATE, and DELETE. It is, however, very powerful, in that with one pass through the data it can carry out all three operations. This can improve performance dramatically.

```
merge into employees e using new_employees n
on (e.employee_id = n.employee_id)
when matched then
update set e.salary=n.salary
when not matched then
insert (employee_id,last_name,salary)
values (n.employee_id,n.last_name,n.salary);
```

The preceding statement uses the contents of a table NEW_EMPLOYEES to update or insert rows in EMPLOYEES. The situation could be that EMPLOYEES is a table of all staff, and NEW_EMPLOYEES is a table with rows for new staff and for salary changes for existing staff. The command will pass through NEW_EMPLOYEES, and for each row, attempt to find a row in EMPLOYEES with the same EMPLOYEE_ID. If there is a row found, its SALARY column will be updated with the value of the row in NEW_EMPLOYEES. If there is not such a row, one will be inserted. Variations on the syntax allow the use of a subquery to select the source rows, and it is even possible to delete matching rows.

Control Transactions

The concepts behind a *transaction* are a part of the relational database paradigm. A transaction consists of one or more DML statements, followed by either a ROLLBACK or a COMMIT command. It is possible use the SAVEPOINT.

Before going into the syntax, it is necessary to review the concept of a transaction. Related topics are read consistency; this is automatically implemented by the Oracle server, but to a certain extent programmers can manage it by the way they use the SELECT statement.

Database Transactions

Following is a quick review of some of the principles of a relational database to which all databases (not just Oracle’s) must conform. Other database vendors comply with the same standards with their own mechanisms, but with varying levels of effectiveness. In brief, any relational database must be able to pass the ACID test: it must guarantee atomicity, consistency, isolation, and durability.

A is for Atomicity

The principle of *atomicity* states that all parts of a transaction must complete or none of them. (The reasoning behind the term is that an atom cannot be split—now well known to be a false assumption). For example, if your business analysts have said that every time you change an employee’s salary you must also change the employee’s grade, then the atomic transaction will consist of two updates. The database must guarantee that both go through or neither. If only one of the updates were to succeed, you would have an employee on a salary that was incompatible with his grade: a data corruption, in business terms. If anything (anything at all!) goes wrong before the transaction is complete, the database itself must guarantee that any parts that did go through are reversed; this must happen automatically. But although an atomic transaction sounds small (like an atom), it can be enormous. To take another example, it is logically impossible for an accounting suite nominal ledger to be half in August and half in September: the end-of-month rollover is therefore (in business terms) one atomic transaction, which may affect millions of rows in thousands of tables and take hours to complete (or to roll back, if anything goes wrong). The rollback of an incomplete transaction may be manual (as when you issue the ROLLBACK command), but it must be automatic and unstoppable in the case of an error.

C is for Consistency

The principle of *consistency* states that the results of a query must be consistent with the tate of the database at the time the query started. Imagine a simple query that averages the value of a column of a table. If the table is large, it will take many minutes to pass through the table. If other users are updating the column while the query is in progress, should the query include the new or the old values? Should it include rows that were inserted or deleted after the query started? The principle of consistency requires that the database ensure that changed values are not seen by the query; it will give you an average of the column as it was when the query started, no matter how long the query takes or what other activity is occurring on the tables concerned. Oracle guarantees that if a query succeeds, the result will be consistent. However, if the database administrator has not configured the database appropriately, the query may not succeed: there is a famous Oracle error, “ORA-1555 snapshot too old,” that is raised. This used to be an extremely difficult problem to fix with earlier releases of the database, but with recent versions the database administrator should always be able to prevent this.

I is for Isolation

The principle of *isolation* states that an incomplete (that is, uncommitted) transaction must be invisible to the rest of the world. While the transaction is in progress, only the one session that is executing the transaction is allowed to see the

changes; all other sessions must see the unchanged data, not the new values. The logic behind this is first, that the full transaction might not go through (remember the principle of atomicity and automatic or manual rollback?) and that therefore no other users should be allowed to see changes that might be reversed. And second, during the progress of a transaction the data is (in business terms) incoherent: there is a short time when the employee has had his salary changed but not his grade. Transaction isolation requires that the database must conceal transactions in progress from other users: they will see the preupdate version of the data until the transaction completes, when they will see all the changes as a consistent set. Oracle guarantees transaction isolation: there is no way any session (other than that making the changes) can see uncommitted data. A read of uncommitted data is known as a *dirty read*, which Oracle does not permit (though some other databases do).

D is for Durable

The principle of *durability* states that once a transaction completes, it must be impossible for the database to lose it. During the time that the transaction is in progress, the principle of isolation requires that no one (other than the session concerned) can see the changes it has made so far. But the instant the transaction completes, it must be broadcast to the world, and the database must guarantee that the change is never lost; a relational database is not allowed to lose data. Oracle fulfills this requirement by writing out all change vectors that are applied to data to log files as the changes are done. By applying this log of changes to backups taken earlier, it is possible to repeat any work done in the event of the database being damaged. Of course, data can be lost through user error such as inappropriate DML, or dropping or truncating tables. But as far as Oracle and the DBA are concerned, such events are transactions like any other: according to the principle of durability, they are absolutely nonreversible.

The Start and End of a Transaction

A session begins a transaction the moment it issues any INSERT, UPDATE, or DELETE statement (but not a TRUNCATE—that is a DDL command, not DML).

The transaction continues through any number of further DML commands until the session issues either a COMMIT or a ROLLBACK statement. Only then will the changes be made permanent and become visible to other sessions (if it is committed, rather than rolled back). It is impossible to nest transactions. The SQL standard does not allow a user to start one transaction and then start another before terminating the first. This can be done with PL/SQL (Oracle's proprietary third generation language), but not with industry-standard SQL.

The explicit transaction control statements are COMMIT, ROLLBACK, and SAVEPOINT. There are also circumstances other than a user-issued COMMIT or ROLLBACK that will implicitly terminate a transaction:

- Issuing a DDL or DCL statement
- Exiting from the user tool (SQL*Plus or SQL Developer or anything else)
- If the client session dies
- If the system crashes

If a user issues a DDL (CREATE, ALTER, or DROP) or DCL (GRANT or REVOKE) command, the transaction he has in progress (if any) will be committed:

it will be made permanent and become visible to all other users. This is because the DDL and DCL commands are themselves transactions. If it were possible to see the source code for these commands, it would be obvious. They adjust the data structures by performing DML commands against the tables that make up the data dictionary, and these commands are terminated with a COMMIT. If they were not, the changes made couldn't be guaranteed to be permanent. As it is not possible in SQL to nest transactions, if the user already has a transaction running, the statements the user has run will be committed along with the statements that make up the DDL or DCL command.

If a user starts a transaction by issuing a DML command and then exits from the tool he is using without explicitly issuing either a COMMIT or a ROLLBACK, the transaction will terminate—but whether it terminates with a COMMIT or

a ROLLBACK is entirely dependent on how the tool is written. Many tools will have different behavior, depending on how the tool is exited. (For instance, in the Microsoft Windows environment, it is common to be able to terminate a program either by selecting the File | Exit options from a menu on the top left of the window, or by clicking an "X" in the top right corner. The programmers who wrote the tool may well have coded different logic into these functions.) In either case, it will be a controlled exit, so the programmers should issue either a COMMIT or a ROLLBACK, but the choice is up to them.

If a client's session fails for some reason, the database will always roll back the transaction. Such failure could be for a number of reasons: the user process can die or be killed at the operating system level, the network connection to the database server may go down, or the machine where the client tool is running can crash.

In any of these cases, there is no orderly issue of a COMMIT or ROLLBACK statement, and it is up to the database to detect what has happened. The behavior is that the session is killed, and an active transaction is rolled back. The behavior is the same if the failure is on the server side. If the database server crashes for any reason, when it next starts up all transactions from any sessions that were in progress will be rolled back.

The Transaction Control Statements

A transaction begins implicitly with the first DML statement. There is no command to explicitly start a transaction. The transaction continues through all subsequent DML statements issued by the session. These statements can be against any number of tables: a transaction is not restricted to one table. It terminates (barring any of the events listed in the previous section) when the session issues a COMMIT or ROLLBACK command. The SAVEPOINT command can be used to set markers that will stage the action of a ROLLBACK, but the same transaction remains in progress irrespective of the use of SAVEPOINT.

COMMIT

Syntactically, COMMIT is the simplest SQL command. The syntax is as follows:

```
COMMIT;
```

This will end the current transaction, which has the dual effect of making the changes both permanent and visible to other sessions. Until a transaction is committed, it cannot be seen by any other sessions, even if they are logged on to the database with the same username as that of the session executing the transactions. Until a transaction is committed, it is invisible to other sessions and can be reversed. But once it is committed, it is absolutely nonreversible. The principle of durability applies.

The state of the data before the COMMIT is that the changes have been made, but all sessions other than the one that made the changes are redirected to copies of the data in its prechanged form. So if the session has inserted rows, other sessions that SELECT from the table will not see them. If the transaction has deleted rows, other sessions selecting from the table will still see them. If the transaction has made updates, it will be the unupdated versions of the rows that are presented to other sessions. This is in accordance with the principle of isolation: no session can be in any way dependent on the state of an uncommitted transaction.

After the COMMIT, all sessions will immediately see the new data in any queries they issue: they will see the new rows, they will not see the deleted rows, they will see the new versions of the updated rows. In the case of the hypothetical conversation just discussed, one moment other sessions' queries will see millions of rows in the table; the next it will be empty. This is in accordance with the principle of durability.

ROLLBACK

While a transaction is in progress, Oracle keeps an image of the data as it was before the transaction. This image is presented to other sessions that query the data while the transaction is in progress. It is also used to roll back the transaction automatically if anything goes wrong, or deliberately if the session requests it. The syntax to request a rollback is as follows:

```
ROLLBACK [TO SAVEPOINT savepoint] ;
```

The optional use of savepoints is detailed in the section following. The state of the data before the rollback is that the data has been changed, but the information needed to reverse the changes is available. This information is presented to all other sessions, in order to implement the principle of isolation. The rollback will discard all the changes by restoring the prechange image of the data; any rows the transaction inserted will be deleted, rows the transaction deleted will be inserted back into the table, and any rows that were updated will be returned to their original state. Other sessions will not be aware that anything has happened at all; they never saw the changes. The session that did the transaction will now see the data as it was before the transaction started.

A COMMIT is instantaneous, because it doesn't really have to do anything. The work has already been done. A ROLLBACK can be very slow: it will usually take as long (if not longer) to reverse a transaction than it took to make the changes in the first place. Rollbacks are not good for database performance.

Use the COMMIT and ROLLBACK Commands

In this exercise, demonstrate the use of transaction control statements and transaction isolation. It is assumed that the HR.REGIONS table is as seen in the illustration at the end of Exercise 9-3. If not, adjust the values as necessary. Connect to the HR schema with two sessions concurrently. These can be two SQL*Plus sessions or two SQL Developer sessions or one of each. The following table lists steps to follow in each session.

1 `select * from regions; select * from regions;`

Both sessions see the same data.

2 `insert into regions values(100,'UK');`

`insert into regions values(101,'GB');`

3 `select * from regions; select * from regions;`

Both sessions see different results: the original data, plus their own change.

4 `commit;`

5 `select * from regions; select * from regions;`

One transaction has been published to the world, the other is still visible to only one session.

6 `rollback; rollback;`

7 `select * from regions; select * from regions;`

The committed transaction was not reversed because it has already been committed, but the uncommitted one is now completely gone, having been terminated by rolling back the change.

8 `delete from regions where region_id=100;`

`delete from regions where region_id=101;`

9 `select * from regions; select * from regions;`

Each deleted row is still visible in the session that did not delete it, until you do the following:

10 `commit; commit;`

11 `select * from regions; select * from regions;`

With all transactions terminated, both sessions see a consistent view of the table.

SAVEPOINT

The use of savepoints is to allow a programmer to set a marker in a transaction that can be used to control the effect of the ROLLBACK command. Rather than rolling back the whole transaction and terminating it, it becomes possible to reverse all changes made after a particular point but leave changes made before that point intact. The transaction itself remains in progress: still uncommitted, still rollbackable, and still invisible to other sessions.

The syntax is as follows:

SAVEPOINT *savepoint*;

This creates a named point in the transaction that can be used in a subsequent ROLLBACK command. The following table illustrates the number of rows in a table at various stages in a transaction. The table is a very simple table called TAB, with one column.

The example in the table shows two transactions: the first terminated with a COMMIT, the second with a ROLLBACK. It can be seen that the use of savepoints is visible only within the transaction: other sessions see nothing that is not committed.

The SAVEPOINT command is not (yet) part of the official SQL standard, so it may be considered good practice to avoid it in production systems. It can be very useful in development, though, when you are testing the effect of DML statements and walking through a complex transaction step by step.

The AUTOCOMMIT in SQL*Plus and SQL Developer

The standard behavior of SQL*Plus and SQL Developer is to follow the SQL standard: a transaction begins implicitly with a DML statement and ends explicitly with a COMMIT or a ROLLBACK. It is possible to change this behavior in both tools so that every DML statement commits immediately, in its own transaction. If this is done, there is no need for any COMMIT statements, and the ROLLBACK statement can never have any effect: all DML statements become permanent and visible to others as soon as they execute.

```
truncate table tab; 0 0
insert into tab values ('one'); 1 0
savepoint first; 1 0
insert into tab values ('two'); 2 0
savepoint second; 2 0
insert into tab values ('three'); 3 0
rollback to savepoint second; 2 0
rollback to savepoint first; 1 0
commit; 1 1
delete from tab; 0 1
rollback; 1 1
```

In SQL*Plus, enable the autocommit mode with the command:

SET AUTOCOMMIT ON

To return to normal:

SET AUTOCOMMIT OFF

In SQL Developer, from the Tools menu, select Preferences. Then expand Database and Worksheet Parameters: you will see the Autocommit in SQL Worksheet check box.

It may be hard to justify enabling the autocommit mode of the SQL*Plus and SQL Developer tools. Perhaps the only reason is for compatibility with some third-party products that do not follow the SQL standard. SQL scripts written for such products may not have any COMMIT statements.

SELECT FOR UPDATE

One last transaction control statement is SELECT FOR UPDATE. Oracle, by default, provides the highest possible level of concurrency: readers do not block writers, and writers do not block readers. Or in plain language, there is no problem with one session querying data that another session is updating, or one session updating data that another session is querying. However, there are times when you may wish to change this behavior and prevent changes to data that is being queried.

It is not unusual for an application to retrieve a set of rows with a SELECT command, present them to a user for perusal, and prompt him for any changes. Because Oracle is a multiuser database, it is not impossible that another session has also retrieved the same rows. If both sessions attempt to make changes, there can be some rather odd effects. The following table depicts such a situation.

```
select * from regions; select * from regions;
delete from regions
where region_id=5;
commit;
update regions set region_name='GB'
where region_id=5;
```

This is what the first user will see, from a SQL*Plus prompt:

```
SQL> select * from regions;
REGION_ID REGION_NAME
---------- -------------------------
5 UK
1 Europe
2 Americas
3 Asia
4 Middle East and Africa
SQL> update regions set region_name='GB' where region_id=5;
0 rows updated.
```

This is a bit disconcerting. One way around this problem is to lock the rows in which one is interested:

```
select * from regions for update;
```

Understanding Transaction Isolation

All DML statements are private to the session that makes them, until the transaction commits. The transaction is started implicitly with the first DML statement executed. Until it is committed, it can be reversed with a ROLLBACK. No other session will ever see changes that have not been committed, but the instant they are committed they will be visible to all other sessions.

Transaction structure is vital for good programming. A transaction is a logical unit of work: the changes made by the transaction, whether it is one statement affecting one row n one table, or many statements affecting any number of rows in many tables, should be self contained. It should not be in any way dependent on statements executed outside the transaction, and it should not be divisible into smaller, self-contained transactions. A transaction should be the right size; it should contain all the statements that cannot be separated in terms of business logic and no statements that can be.

The decisions on transaction structure may be complex, but with some thought and investigation, they can be made for any situation. Business and systems analysts can and should advise.

The FOR UPDATE clause will place a lock on all the rows retrieved. No changes can be made to them by any session other than that which issued the command, and therefore the subsequent updates will succeed: it is not possible for the rows to have been changed. This means that one session will have a consistent view of the data (it won't change), but the price to be paid is that other sessions will hang if they try to update any of the locked rows (they can, of course, query them). The locks placed by a FOR UPDATE clause will be held until the session issuing the command issues a COMMIT or ROLLBACK. This must be done to release the locks, even if no DML commands have been executed.

TWO-MINUTE DRILL

Describe Each Data Manipulation Language (DML) Statement

❑ INSERT enters rows into a table.

❑ UPDATE adjusts the values in existing rows.

❑ DELETE removes rows.

❑ MERGE can combine the functions of INSERT, UPDATE, and DELETE.

❑ Even though TRUNCATE is not DML, it does remove all rows in a table.

Insert Rows into a Table

❑ INSERT can enter one row or a set of rows.

❑ It is possible for an INSERT to enter rows into multiple tables.

❑ Subqueries can be used to generate the rows to be inserted.

❑ Subqueries and functions can be used to generate column values.

❑ An INSERT is not permanent until it is committed.

Update Rows in a Table

❑ UPDATE can affect one row or a set of rows in one table.

❑ Subqueries can be used to select the rows to be updated.

❑ Subqueries and functions can be used to generate column values.

❑ An UPDATE is not permanent until it is committed.

Delete Rows from a Table

❑ DELETE can remove one row or a set of rows from one table.

❑ A subquery can be used to select the rows to be deleted.

❑ A DELETE is not permanent until it is committed.

❑ TRUNCATE removes every row from a table.

❑ A TRUNCATE is immediately permanent: it cannot be rolled back.

Control Transactions

❑ A transaction is a logical unit of work, possibly several DML statements.

❑ Transactions are invisible to other sessions until committed.

❑ Until committed, transactions can be rolled back.

❑ Once committed, a transaction cannot be reversed.

❑ A SAVEPOINT lets a session roll back part of a transaction.

SELF TEST

Choose all the correct answers for each question.

Describe Each Data Manipulation Language (DML) Statement

1. Which of the following commands can be rolled back?
A. COMMIT
B. DELETE
C. INSERT
D. MERGE
E. TRUNCATE
F. UPDATE

2. How can you change the primary key value of a row? (Choose the best answer.)
A. You cannot change the primary key value.
B. Change it with a simple UPDATE statement.
C. The row must be removed with a DELETE and reentered with an INSERT.
D. This is only possible if the row is first locked with a SELECT FOR UPDATE.

3. If an UPDATE or DELETE command has a WHERE clause that gives it a scope of several rows, what will happen if there is an error part way through execution? The command is one of several in a multistatement transaction. (Choose the best answer.)
A. The command will skip the row that caused the error and continue.
B. The command will stop at the error, and the rows that have been updated or deleted will remain updated or deleted.
C. Whatever work the command had done before hitting the error will be rolled back, but work done already by the transaction will remain.
D. The whole transaction will be rolled back.

Insert Rows into a Table

4. If a table T1 has four numeric columns, C1, C2, C3, and C4, which of these statements will succeed? (Choose the best answer.)
A. insert into T1 values (1,2,3,null);
B. insert into T1 values (‘1’,‘2’,‘3’,‘4’);
C. insert into T1 select * from T1;
D. All the statements (A, B, and C) will succeed.
E. None of the statements (A, B, or C) will succeed.

5. Study the result of this SELECT statement:

```
SQL> select * from t1;
C1 C2 C3 C4
---------- ---------- ---------- ----------
1 2 3 4
5 6 7 8
```

If you issue this statement:

```
insert into t1 (c1,c2) values(select c1,c2 from t1);
```

why will it fail? (Choose the best answer.)
A. Because values are not provided for all the table’s columns: there should be NULLs for C3 and C4.
B. Because the subquery returns multiple rows: it requires a WHERE clause to restrict the number of rows returned to one.
C. Because the subquery is not scalar: it should use MAX or MIN to generate scalar values.
D. Because the VALUES keyword is not used with a subquery.
E. It will succeed, inserting two rows with NULLs for C3 and C4.

6. Consider this statement:

```
insert into regions (region_id,region_name)
values ((select max(region_id)+1 from regions), 'Great Britain');
```

What will the result be? (Choose the best answer.)
A. The statement will not succeed if the value generated for REGION_ID is not unique, because REGION_ID is the primary key of the REGIONS table.
B. The statement has a syntax error because you cannot use the VALUES keyword with a subquery.
C. The statement will execute without error.
D. The statement will fail if the REGIONS table has a third column.

Update Rows in a Table

7. You want to insert a row and then update it. What sequence of steps should you follow? (Choose the best answer.)
A. INSERT, UPDATE, COMMIT
B. INSERT, COMMIT, UPDATE, COMMIT
C. INSERT, SELECT FOR UPDATE, UPDATE, COMMIT
D. INSERT, COMMIT, SELECT FOR UPDATE, UPDATE, COMMIT

8. If you issue this command:

```
update employees set salary=salary * 1.1;
```

what will be the result? (Choose the best answer.)
A. The statement will fail because there is no WHERE clause to restrict the rows affected.
B. The first row in the table will be updated.
C. There will be an error if any row has its SALARY column NULL.
D. Every row will have SALARY incremented by 10 percent, unless SALARY was NULL.

Delete Rows from a Table

9. How can you delete the values from one column of every row in a table? (Choose the best answer.)
A. Use the DELETE COLUMN command.
B. Use the TRUNCATE COLUMN command.
C. Use the UPDATE command.
D. Use the DROP COLUMN command.

10. Which of these commands will remove every row in a table? (Choose one or more correct answers.)
A. A DELETE command with no WHERE clause
B. A DROP TABLE command
C. A TRUNCATE command
D. An UPDATE command, setting every column to NULL and with no WHERE clause

Control Transactions

11. User JOHN updates some rows and asks user ROOPESH to log in and check the changes before he commits them. Which of the following statements is true? (Choose the best answer.)
A. ROOPESH can see the changes but cannot alter them because JOHN will have locked the rows.

B. ROOPESH will not be able to see the changes.
C. JOHN must commit the changes so that ROOPESH can see them and, if necessary, roll them back.
D. JOHN must commit the changes so that ROOPESH can see them, but only JOHN can roll them back.

12. User JOHN updates some rows but does not commit the changes. User ROOPESH queries the rows that JOHN updated. Which of the following statements is true? (Choose three correct answers.)
A. ROOPESH will not be able to see the rows because they will be locked.
B. ROOPESH will be able to see the new values, but only if he logs in as JOHN.
C. ROOPESH will see the old versions of the rows.
D. ROOPESH will see the state of the state of the data as it was when JOHN last created a SAVEPOINT.

13. Which of these commands will terminate a transaction? (Choose three correct answers.)
A. COMMIT
B. DELETE
C. ROLLBACK
D. ROLLBACK TO SAVEPOINT
E. SAVEPOINT
F. TRUNCATE

LAB QUESTION

Carry out this exercise in the OE schema.

1. Insert a customer into CUSTOMERS, using a function to generate a unique customer number:

```
insert into customers
(customer_id,cust_first_name,cust_last_name)
values((select max(customer_id)+1 from customers),'John','Watson');
```

2. Give him a credit limit equal to the average credit limit:

```
update customers set
credit_limit=(select avg(credit_limit) from customers)
where cust_last_name='Watson';
```

3. Create another customer using the customer just created, but make sure the CUSTOMER_ID is unique:

```
insert into customers
(customer_id,cust_first_name,cust_last_name,credit_limit)
select customer_id+1,cust_first_name,cust_last_name,credit_limit
from customers
where cust_last_name='Watson';
```

4. Change the name of the second entered customer:

```
update customers
set cust_last_name='Ramklass',cust_first_name='Roopesh'
where customer_id=(select max(customer_id) from customers);
```

5. Commit this transaction:

```
commit;
```

6. Determine the CUSTOMER_IDs of the two new customers and lock the rows:

7. select customer_id,cust_last_name from customers where cust_last_name in (‘Watson’,‘Ramklass’) for update;
From another session connected to the OE schema, attempt to update one of the locked rows:

```
update customers set credit_limit=0 where cust_last_name='Ramklass';
```

8. This command will hang. In the first session, release the locks by issuing a commit:

```
commit;
```

9. The second session will now complete its update. In the second session, delete the two rows:

```
delete from customers where cust_last_name in ('Watson','Ramklass');
```

10. In the first session, attempt to truncate the CUSTOMERS table:

```
truncate table customers;
```

11. This will fail because there is a transaction in progress against the table, which will block all DDL commands. In the second session, commit the transaction:

```
commit;
```

12. The CUSTOMERS table will now be back in the state it was in at the start of the exercise. Confirm this by checking the value of the highest CUSTOMER_ID:

```
select max(customer_id) from customers;
```

SELF TEST ANSWERS

Describe Each Data Manipulation Language (DML) Statement

1. ✹✓ B, C, D, F. These are the DML commands: they can all be rolled back.

✹⋎ **A, E.** COMMIT terminates a transaction, which can then never be rolled back. TRUNCATE is a DDL command and includes a built-in COMMIT.

2. ✹✓ B. Assuming no constraint violations, the primary key can updated like any other column.

✹⋎ **A, C, D. A** is wrong because there is no restriction on updating primary keys (other than constraints). **C** is wrong because there is no need to do it in such a complex manner. **D** is wrong because the UPDATE will apply its own lock: you do not have to lock the row first.

3. ✹✓ C. This is the expected behavior: the statement is rolled back, and the rest of the transaction remains uncommitted.

✹⋎ **A, B, D. A** is wrong because, while this behavior is in fact configurable, it is not enabled by default. **B** is wrong because, while this is in fact possible in the event of space errors, it is not enabled by default. **D** is wrong because only the one statement will be rolled back, not the whole transaction.

Insert Rows into a Table

4. ✹✓ D. A, B, and **C** will all succeed, even though **B** will force the database to do some automatic type casting.

✹⋎ **A, B, C, E. A, B,** and **C** are wrong because each one will succeed. **E** is wrong because **A**, **B**, and **C** will all succeed.

5. ✹✓ D. The syntax is wrong: use either the VALUES keyword or a subquery, but not both. Remove the VALUES keyword, and it will run. C3 and C4 would be populated with NULLs.

✹⋎ **A, B, C, E. A** is wrong because there is no need to provide values for columns not listed. **B** and **C** are wrong because an INSERT can insert a set of rows, so there is no need to restrict the number with a WHERE clause or by using MAX or MIN to return only one row. **E** is wrong because the statement is not syntactically correct.

6. ✹✓ C. The statement is syntactically correct, and the use of “MAX(REGION_ID) + 1” guarantees generating a unique number for the primary key column.

✹⋎ **A, B, D. A** is wrong because the function will generate a unique value for the primary key. **B** is wrong because there is no problem using a scalar subquery to generate a value for a VALUES list. What cannot be done is to use the VALUES keyword and then a single nonscalar subquery to provide all the values. **D** is wrong because if there is a third column, it will be populated with a NULL value.

Update Rows in a Table

7. ✹✓ A. This is the simplest (and therefore the best) way.

✹✗ **B, C, D.** All these will work, but they are all needlessly complicated: no programmer should use unnecessary statements.

8. ✹✓ D. Any arithmetic operation on a NULL returns a NULL, but all other rows will be updated.

✹✗ **A, B, C. A** and **B** are wrong because the lack of a WHERE clause means that every row will be processed. **C** is wrong because trying to do arithmetic against a NULL is not an error (though it isn't very useful, either).

Delete Rows from a Table

9. ✹✓ C. An UPDATE, without a WHERE clause, is the only way.

✹✗ **A, B, D. A** is wrong because there is no such syntax: a DELETE affects the whole row. B is wrong because there is no such syntax: a TRUNCATE affects the whole table. **D** is wrong because, while this command does exist (it is part of the ALTER TABLE command), it will remove the column completely, not just clear the values out of it.

10. ✹✓ A, C. The TRUNCATE will be faster, but the DELETE will get there too.

✹✗ **B** is wrong because this will remove the table as well as the rows within it. **D** is wrong because the rows will still be there—even though they are populated with NULLs.

Control Transactions

11. ✹✓ B. The principle of isolation means that only JOHN can see his uncommitted transaction.

✹✗ **A, C, D. A** is wrong because transaction isolation means that no other session will be able to see the changes. **C** and **D** are wrong because a committed transaction can never be rolled back.

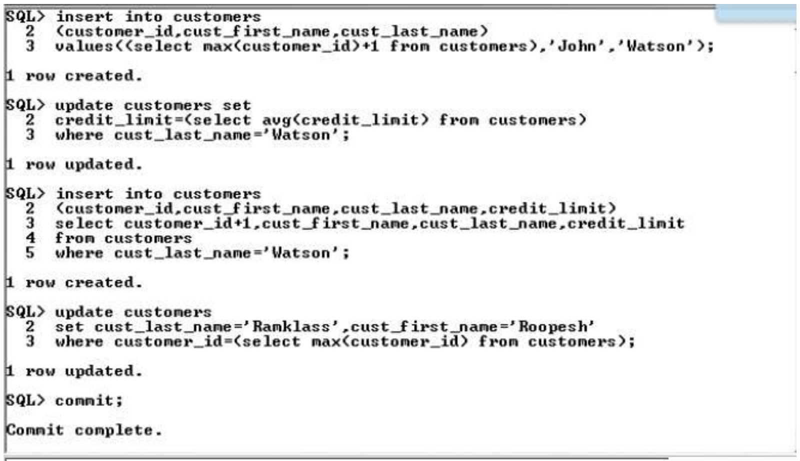
12. ✹✓ C. Transaction isolation means that no other session will be able to see the changes until they are committed.

✹✗ **A, B, D. A** is wrong because locking is not relevant; writers do not block readers. **B** is wrong because isolation restricts visibility of in-progress transactions to the session making the changes; the schema the users are connecting to does not matter. **D** is wrong because savepoints are only markers in a transaction; they do not affect publishing changes to other sessions.

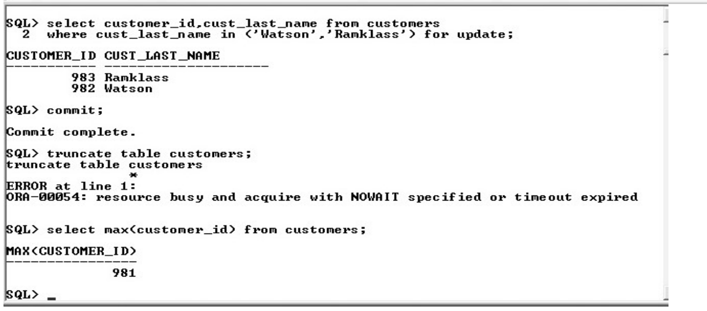
13. ✹✓ A, C, F. COMMIT and ROLLBACK are the commands to terminate a transaction explicitly; TRUNCATE will do it implicitly.

✹✗ **B, D, E. B** is wrong because DELETE is a DML command that can be executed within a transaction. **D** and **E** are wrong because creating savepoints and rolling back to them leave the transaction in progress.

LAB ANSWER



```
SQL> insert into customers
  2  (customer_id,cust_first_name,cust_last_name)
  3  values((select max(customer_id)+1 from customers),'John','Watson');

1 row created.

SQL> update customers set
  2  credit_limit=(select avg(credit_limit) from customers)
  3  where cust_last_name='Watson';

1 row updated.

SQL> insert into customers
  2  (customer_id,cust_first_name,cust_last_name,credit_limit)
  3  select customer_id+1,cust_first_name,cust_last_name,credit_limit
  4  from customers
  5  where cust_last_name='Watson';

1 row created.

SQL> update customers
  2  set cust_last_name='Ramklass',cust_first_name='Roopesh'
  3  where customer_id=(select max(customer_id) from customers);

1 row updated.

SQL> commit;

Commit complete.
```



```
SQL> select customer_id,cust_last_name from customers
  2  where cust_last_name in ('Watson','Ramklass') for update;

CUSTOMER_ID CUST_LAST_NAME
----------- --------------------
        983 Ramklass
        982 Watson

SQL> commit;

Commit complete.

SQL> truncate table customers;
truncate table customers
               *
ERROR at line 1:
ORA-00054: resource busy and acquire with NOWAIT specified or timeout expired


SQL> select max(customer_id) from customers;

MAX(CUSTOMER_ID)
----------------
             981

SQL> _
```

Chapter 11: Using DDL Statements to Create and Manage Tables

There are several types of data objects in a database that can be addressed by users with
SQL. The most commonly used type of object is the table.
When creating a table, there are certain rules that must be followed regarding the table's structure: its columns may only be of certain data types. There are also rules that can be defined for the individual rows; these are known as *constraints*.
The structural rules and the constraint rules together restrict the data that can be inserted into the table.

Categorize the Main Database Objects

There are various types of objects that can exist within a database, many more with the current release than with earlier versions. All objects have a names, and all objects are owned by someone. The "someone" is a database user, such as HR. The objects the user owns are their *schema*. An object's name must conform to certain rules.
The objects of greatest interest to a SQL programmer are those that contain, or give access to, data. These are

- Tables
- Views
- Synonyms
- Indexes
- Sequences

Users and Schemas

A *user* is a person who can connect to the database. The user will have a username and a password. A *schema* is a container for the objects owned by a user. When a user is created, their schema is created too.
A schema is the objects owned by a user; initially, it will be empty.
Some schemas will always be empty: the user will never create any objects, because they do not need to and (if the user is set up correctly) will not have the necessary privileges anyway.
Users such as this will have been granted permissions, either through direct privileges or through roles, to use code and access data in other schemas, owned by other users. Other users may be the reverse of this: they will own many objects but will never actually log on to the database. They need not even have been granted the CREATE SESSION privilege, so the account is effectively disabled (or indeed it can be locked)—these schemas are used as repositories for code and data accessed by others.
Schema objects are objects with an owner. The unique identifier for an object of a particular type is not its name—it is its name, prefixed with the name of the schema to which it belongs. Thus the table HR.REGIONS is a table called REGIONS, which is owned by user HR. There could be another table SYSTEM.REGIONS that would be a completely different table (perhaps different in both structure and contents) owned by user SYSTEM and residing in his or her schema.
A number of users (and their associated schemas) are created automatically at database creation time. Principal among these are SYS and SYSTEM. User SYS owns the data dictionary: a set of tables (in the SYS schema) that define the database and its contents. SYS also owns several hundred PL/SQL packages: code that is provided for the use of database administrators and developers. Objects in the SYS schema should never be modified with DML commands.

Naming Schema Objects

A schema object is an object that is owned by a user. All schema object names must conform to certain rules:

- The name may be from 1 to 30 characters long.
- Reserved words (such as SELECT) cannot be used as object names.
- All names must begin with a letter from "A" through "Z."
- The characters in a name can only be letters, numbers, an underscore (_), the dollar sign ($), or the hash symbol (#).
- Lowercase letters will be converted to uppercase.

By enclosing the name within double quotes, all these rules (with the exception of the length) can be broken, but to get to the object, subsequently, it must always be specified with double quotes.
Note that the same restrictions also apply to column names.
Although tools such as SQL*Plus and SQL Developer will automatically convert lowercase letters to uppercase unless the name is enclosed within double quotes;
remember that object names are always case sensitive. In this example, the two tables are completely different:

```
SQL> create table lower(c1 date);
Table created.
SQL> create table "lower"(col1 varchar2(2));
Table created.
SQL> select table_name from dba_tables where lower(table_name) = 'lower';
TABLE_NAME
------------------------------
lower
LOWER
```

While it is possible to use lowercase names and nonstandard characters (even spaces), it is considered bad practice because of the confusion it can cause.

Object Namespaces

It is often said that the unique identifier for an object is the object name, prefixed with the schema name. While this is generally true, for a full understanding of naming, it is necessary to introduce the concept of a *namespace*. A namespace defines a group of object types, within which all names must be uniquely identified, by schema and name. Objects in different namespaces can share the same name.
These object types all share the same namespace:

- Tables
- Views
- Sequences
- Private synonyms

Thus it is impossible to create a view with the same name as a table—at least, it is impossible if they are in the same schema. And once created, SQL statements can address a view or a synonym as though it were a table. The fact that tables, views, and private synonyms share the same namespace means that you can set up several layers of abstraction between what the users see and the actual tables, which can be invaluable for both security and for simplifying application development. Indexes and constraints each have their own namespace. Thus it is possible for an index to have the same name as a table, even within the same schema.

Determine What Objects Are Accessible to Your Session

In this exercise, query various data dictionary views as user HR to determine what objects are in the HR schema and what objects in other schemas HR has access to.

1. Connect to the database with SQL*Plus or SQL Developer as user HR.
2. Determine how many objects of each type are in the HR schema:

```
select object_type,count(*) from user_objects group by object_type;
```

The USER_OBJECTS view lists all objects owned by the schema to which the current session is connected, in this case HR.

3. Determine how many objects in total HR has permissions on:

```
select object_type,count(*) from all_objects group by object_type;
```

The ALL_OBJECTS view lists all objects to which the user has some sort of access.

4. Determine who owns the objects HR can see:

```
select distinct owner from all_objects;
```

Review the Table Structure

According to the relational database paradigm, a table is a two-dimensional structure storing rows. A row is one or more columns. Every row in the table has the same columns, as defined by the structure of the table. The Oracle database does permit variations on this two-dimensional model. Some columns can be defined as nested tables, which themselves have several columns. Other columns may be of an unbounded data type such as a binary large object, theoretically terabytes big. It is also possible to define columns as objects. The object will have an internal structure (possibly based on columns) that is not visible as part of the table. The systems analysis phase of the system development lifecycle will have modeled the data structures needed to store the system's information into third normal form. The result is a set of two-dimensional tables, each with a primary key and linked to each other with foreign keys. The system design phase may have compromised this structure, perhaps by denormalizing the tables or by taking advantage of Oracle-specific capabilities such as nested tables. But the end result, as far as the SQL developer is concerned, is a set of tables. Each table exists as a definition in the data dictionary. On creation, the table will have been assigned a limited amount of space (known as an *extent*) within the database. This may be small, perhaps only a few kilobytes or megabytes. As rows are inserted into the table, this extent will fill. When it is full, the database will (automatically) assign another extent to the table. As rows are deleted, space within the assigned extents becomes available for reuse. Even if every row is deleted, the extents remain allocated to the table. They will only be freed up and returned to the database for use elsewhere if the table is dropped or truncated.

Investigate Table Structures

In this exercise, query various data dictionary views as user HR to determine the structure of a table.

1. Connect to the database with SQL*Plus or SQL Developer as user HR.
2. Determine the names and types of tables that exist in the HR schema:

```
select table_name,cluster_name,iot_type from user_tables;
```

Clustered tables and index organized tables (IOTs) are advanced table structures. In the HR schema, all tables are standard heap tables except for COUNTRIES which is an IOT.

3. Use the DESCRIBE command to display the structure of a table:

```
describe regions;
```

4. Retrieve similar information by querying a data dictionary view:

```
select column_name,data_type,nullable from user_tab_columns where table_name='REGIONS';
```

List the Data Types That Are Available for Columns

When creating tables, each column must be assigned a data type, which determines the nature of the values that can be inserted into the column. These data types are also used to specify the nature of the arguments for PL/SQL procedures and functions.

When selecting a data type, you must consider the data that you need to store and the operations you will want to perform upon it. Space is also a consideration: some data types are fixed length, taking up the same number of bytes no matter what data is actually in it; others are variable. If a column is not populated, then Oracle will not give it any space at all. If you later update the row to populate the column, then the row will get bigger, no matter whether the data type is fixed length or variable.

The following are the data types for alphanumeric data:

- **VARCHAR2** Variable-length character data, from 1 byte to 4KB. The data is stored in the database character set.

For ISO/ANSI compliance, you can specify a VARCHAR data type, but any columns of this type will be automatically converted to VARCHAR2.

- **NVARCHAR2** Like VARCHAR2, but the data is stored in the alternative national language character set, one of the permitted Unicode character sets.
- **CHAR** Fixed-length character data, from 1 byte to 2KB, in the database character set. If the data is not the length of the column, then it will be padded with spaces.

The following is the data type for binary data:

- **RAW** Variable-length binary data, from 1 byte to 4KB. Unlike the CHAR and VARCHAR2 data types, RAW data is not converted by Oracle Net from the database's character set to the user process's character set on SELECT or the other way on INSERT.

The following are the data types for numeric data, all variable length:

- **NUMBER** Numeric data, for which you can specify precision and scale. The precision can range from to 1 to 38, the scale can range from 84 to 127.
- **FLOAT** This is an ANSI data type, floating-point number with precision of 126 binary (or 38 decimal). Oracle also provides BINARY_FLOAT and BINARY_DOUBLE as alternatives.
- **INTEGER** Equivalent to NUMBER, with scale zero.

The following are the data types for date and time data, all fixed length:

- **DATE** This is either length zero, if the column is empty, or 7 bytes. All DATE data includes century, year, month, day, hour, minute, and second. The valid range is from January 1, 4712 BC to December 31, 9999 AD.
- **TIMESTAMP** This is length zero if the column is empty, or up to 11 bytes, depending on the precision specified. Similar to DATE, but with precision of up to 9 decimal places for the seconds, 6 places by default.
- **TIMESTAMP WITH TIMEZONE** Like TIMESTAMP, but the data is stored with a record kept of the time zone to which it refers. The length may be up to 13 bytes, depending on precision. This data type lets Oracle determine the difference between two times by normalizing them to UTC, even if the times are for different time zones.
- **TIMESTAMP WITH LOCAL TIMEZONE** Like TIMESTAMP, but the data is normalized to the database time zone on saving. When retrieved, it is normalized to the time zone of the user process selecting it.

- **INTERVAL YEAR TO MONTH** Used for recording a period in years and months between two DATEs or TIMESTAMPs.
- **INTERVAL DAY TO SECOND** Used for recording a period in days and seconds between two DATEs or TIMESTAMPs.

The following are the large object data types:

- **CLOB** Character data stored in the database character set, size effectively unlimited: 4GB multiplied by the database block size.
- **NCLOB** Like CLOB, but the data is stored in the alternative national language character set, one of the permitted Unicode character sets.
- **BLOB** Like CLOB, but binary data that will not undergo character set conversion by Oracle Net.
- **BFILE** A locator pointing to a file stored on the operating system of the database server. The size of the files is limited to 4GB.
- **LONG** Character data in the database character set, up to 2GB. All the unctionality of LONG (and more) is provided by CLOB; LONGs should not be used in a modern database, and if your database has any columns of this type they should be converted to CLOB. There can only be one LONG column in a table.
- **LONG RAW** Like LONG, but binary data that will not be converted by Oracle Net. Any LONG RAW columns should be converted to BLOBs.

The following is the ROWID data type:

- **ROWID** A value coded in base 64 that is the pointer to the location of a row in a table. Encrypted. Within it is the exact physical address. ROWID is an Oracle proprietary data type, not visible unless specifically selected.

The VARCHAR2 data type must be qualified with a number indicating the maximum length of the column. If a value is inserted into the column that is less than this, it is not a problem: the value will only take up as much space as it needs. If the value is longer than this maximum, the INSERT will fail with an error. If the value is updated to a longer or shorter value, the length of the column (and therefore the row itself) will change accordingly. If is not entered at all or is updated to NULL, then it will take up no space at all.

The NUMBER data type may optionally be qualified with a precision and a scale. The precision sets the maximum number of digits in the number, and the scale is how many of those digits are to the right of the decimal point. If the scale is negative, this has the effect of replacing the last digits of any number inserted with zeros, which do not count toward the number of digits specified for the precision. If the number of digits exceeds the precision, there will be an error; if it is within the precision but outside the scale, the number will be rounded (up or down) to the nearest value within the scale.

The DATE data type always includes century, year, month, day, hour, minute, and second—even if all these elements are not specified at insert time. Year, month, and date must be specified; if the hours, minutes, and seconds are omitted they will default to midnight. Using the TRUNC function on a date also has the effect of setting the hours, minutes, and seconds to midnight.

Investigate the Data Types in the HR schema

In this exercise, find out what data types are used in the tables in the HR schema, using two techniques.

1. Connect to the database as user HR with SQL*Plus or SQL Developer.
2. Use the DESCRIBE command to show the data types in some tables:

```
describe employees;
describe departments;
```

3. Use a query against a data dictionary view to show what columns make up the EMPLOYEES table, as the DESCRIBE command would:

```
select column_name,data_type,nullable,data_length,data_precision,data_scale
from user_tab_columns where table_name='EMPLOYEES';
```

The view USER_TAB_COLUMNS shows the detail of every column in every table in the current user's schema.

Create a Simple Table

Tables can be stored in the database in several ways. The simplest is the *heap* table. A heap is variable length rows in random order. There may be some correlation between the order in which rows are entered and the order in which they are stored, but this is a matter of luck. More advanced table structures, such as the following, may impose ordering and grouping on the rows or force a random distribution:

- **Index organized tables** Store rows in the order of an index key.
- **Index clusters** Can denormalize tables in parent-child relationships so that related rows from different table are stored together.
- **Hash clusters** Force a random distribution of rows, which will break down any ordering based on the entry sequence.
- **Partitioned tables** Store rows in separate physical structures, the partitions, allocating rows according to the value of a column.

Using the more advanced table structures has no effect whatsoever on SQL. Every SQL statement executed against tables defined with these options will return exactly the same results as though the tables were standard heap tables, so use of these features will not affect code. But while their use is transparent to programmers, they do give enormous benefits in performance.

Creating Tables with Column Specifications

To create a standard heap table, use this syntax:

```
CREATE TABLE [schema.]table [ORGANIZATION HEAP]
(column datatype [DEFAULT expression]
[,column datatype [DEFAULT expression]…);
```

As a minimum, specify the table name (it will be created in your own schema, if you don't specify someone else's) and at least one column with a data type. There are very few developers who ever specify ORGANIZATION HEAP, as this is the default and is industry standard SQL. The DEFAULT keyword in a column definition lets you provide an expression that will generate a value for the column when a row is inserted if a value is not provided by the INSERT statement.

Consider this statement:

```
CREATE TABLE SCOTT.EMP
(EMPNO NUMBER(4),
ENAME VARCHAR2(10),
HIREDATE DATE DEFAULT TRUNC(SYSDATE),
SAL NUMBER(7,2),
COMM NUMBER(7,2) DEFAULT 0.03);
```

This will create a table called EMP in the SCOTT schema. Either user SCOTT himself has to issue the statement (in which case nominating the schema would not actually be necessary), or another user could issue it if he has been granted permission to create tables in another user's schema. Taking the columns one by one:

- EMPNO can be 4 digits long, with no decimal places. If any decimals are included in an INSERT statement, they will be rounded (up or down) to the nearest integer.
- ENAME can store any characters at all, up to ten of them.
- HIREDATE will accept any date, optionally with the time, but if a value is not provided, today's date will be entered as at midnight.
- SAL, intended for the employee's salary, will accept numeric values with up to 7 digits. If any digits over 7 are to the right of the decimal point, they will be rounded off.
- COMM (for commission percentage) has a default value of 0.03, which will be entered if the INSERT statement does not include a value for this column.

Following creation of the table, these statements insert a row and select the result:

```
SQL> insert into scott.emp(empno,ename,sal) values(1000,'John',1000.789);
1 row created.
SQL> select * from emp;
EMPNO ENAME HIREDATE SAL COMM
---------- ---------- --------- ---------- ----------
1000 John 19-NOV-07 1000.79 .03
```

Note that values for the columns not mentioned in the INSERT statement have been generated by the DEFAULT clauses. Had those clauses not been defined in the table definition, the columns would have been NULL. Also note the rounding of the value provided for SAL.

The DEFAULT clause can be useful, but it is of limited functionality. You cannot use a subquery to generate the default value: you can only specify literal values or functions.

Creating Tables from Subqueries

Rather than creating a table from nothing and then inserting rows into it, tables can be created from other tables by using a subquery. This technique lets you create the table definition and populate the table with rows with just one statement. Any query at all can be used as the source of both the table structure and the rows.

The syntax is as follows:

CREATE TABLE [*schema*.]*table* AS *subquery*;

All queries return a two-dimensional set of rows; this result is stored as the new table.

A simple example of creating a table with a subquery is:

```
create table employees_copy as select * from employees;
```

This statement will create a table EMPLOYEES_COPY, which is an exact copy of the EMPLOYEES table, identical in both definition and the rows it contains. Any not null and check constraints on the columns will also be applied to the new table, but any primary-key, unique, or foreign-key constraints will not be. This is because these three types of constraints require indexes that might not be available or desired.

The following is a more complex example:

```
create table emp_dept as select
last_name ename,department_name dname,round(sysdate - hire_date) service
from employees natural join departments order by dname,ename;
```

The rows in the new table will be the result of joining the two source tables, with two of the selected columns having their names changed. The new SERVICE column will be populated with the result of the arithmetic that computes the number of days since the employee was hired. The rows will be inserted in the order specified. This ordering will not be maintained by subsequent DML but, assuming the standard HR schema data, the new table will look like this:

```
SQL> select * from emp_dept where rownum < 10;
ENAME DNAME SERVICE
--------------- --------------- ----------
Gietz Accounting 4914
De Haan Executive 5424
Kochhar Executive 6634
Chen Finance 3705
Faviet Finance 4844
Popp Finance 2905
Sciarra Finance 3703
Urman Finance 3545
Austin IT 3800
9 rows selected.
```

The subquery can of course include a WHERE clause to restrict the rows inserted into the new table. To create a table with no rows, use a WHERE clause that will exclude all rows:

```
create table no_emps as select * from employees where 1=2;
```

The WHERE clause 1=2 can never return TRUE, so the table structure will be created ready for use, but no rows will be inserted at creation time.

Altering Table Definitions after Creation

There are many alterations that can be made to a table after creation. Those that affect the physical storage fall into the domain of the database administrator, but many changes are purely logical and will be carried out by the SQL developers.

The following are examples (for the most part self-explanatory):

- Adding columns:

```
alter table emp add (job_id number);
```

- Modifying columns:

```
alter table emp modify (comm number(4,2) default 0.05);
```

- Dropping columns:

```
alter table emp drop column comm;
```

- Marking columns as unused:

```
alter table emp set unused column job_id;
```

- Renaming columns:

```
alter table emp rename column hiredate to recruited;
```

- Marking the table as read-only:

```
alter table emp read only;
```

All of these changes are DDL commands with the built-in COMMIT. They are therefore nonreversible and will fail if there is an active transaction against the table. They are also virtually instantaneous with the exception of dropping a column. Dropping a column can be a time-consuming exercise because as each column is dropped, every row must be restructured to remove the column's data.

The SET UNUSED command, which makes columns nonexistent as far as SQL is concerned, is often a better alternative, followed when convenient by

```
ALTER TABLE tablename DROP UNUSED COLUMNS;
```

which will drop all the unused columns in one pass through the table.

Marking a table as read-only will cause errors for any attempted DML commands. But the table can still be dropped. This can be disconcerting but is perfectly logical when you think it through. A DROP command doesn't actually affect the table: it affects the tables in the data dictionary that define the table, and these are not read-only.

Dropping and Truncating Tables

Truncate has the effect of removing every row from a table, while leaving the table definition intact. DROP TABLE is more drastic in that the table definition is removed as well.

The syntax is as follows:

DROP TABLE [*schema*.]*tablename* ;

If *schema* is not specified, then the table called *tablename* in your currently logged on schema will be dropped.

As with a TRUNCATE, SQL will not produce a warning before the table is dropped, and furthermore, as with any DDL command, it includes a COMMIT.

A DROP is therefore absolutely nonreversible. But there are some restrictions: if any session (even your own) has a transaction in progress that includes a row in the table, then the DROP will fail, and it is also impossible to drop a table that is referred to in a foreign key constraint defined for a another table. This table (or the constraint) must be dropped first.

Create Tables

In this exercise, use SQL Developer to create a heap table, insert some rows with a subquery, and modify the table. Do some more modifications with SQL*Plus, then drop the table.

1. Connect to the database as user HR with SQL Developer.
2. Right-click the Tables branch of the navigation tree, and click New Table.
3. Name the new table EMPS, and use the Add Column button to set it up as in the following illustration:
4. Click the DDL tab to see if the statement that has been constructed. It should look like this:

```
CREATE TABLE EMPS
(
EMPNO NUMBER,
ENAME VARCHAR2(25),
SALARY NUMBER,
DEPTNO NUMBER(4, 0)
)
;
```

Return to the Table tab (as in the preceding illustration) and click OK to create the table.

5. Run this statement:

```
insert into emps select employee_id,last_name,salary,department_id from employees;
```

and commit the insert:

```
commit;
```

6. Right-click the EMPS table in the SQL Developer navigator, click Column and Add.
7. Define a new column HIRED, type DATE, as in the following illustration below; and click Apply to create the column.
8. Connect to the database as HR with SQL*Plus.
9. Define a default for HIRED column in the EMPS table:

```
alter table emps modify (hired default sysdate);
```

10. Insert a row without specifying a value for HIRED and check that the new row does have a HIRED date but that the other rows do not:

```
insert into emps (empno,ename) values(99,'Newman');
select hired,count(1) from emps group by hired;
```

11. Tidy up by dropping the new table:

```
drop table emps;
```

Explain How Constraints Are Created at the Time of Table Creation

Table constraints are a means by which the database can enforce business rules, and guarantee that the data conforms to the entity-relationship model determined by the systems analysis that defines the application data structures.

For example, the business analysts of your organization may have decided that every customer and every invoice must be uniquely identifiable by number, that no invoices can be issued to a customer before that customer has been created, and that every invoice must have a valid date and a value greater than zero. These would implemented by creating primary-key constraints on the CUSTOMER_NUMBER column of the CUSTOMERS table and the INVOICE_NUMBER column of the INVOICES table, a foreign-key constraint on the INVOICES table referencing the CUSTOMERS table, a not-null constraint on the DATE column of the INVOICES table (the DATE data type will itself ensure that that any dates are valid automatically—it will not accept invalid dates), and a check constraint on the AMOUNT column on the INVOICES table.

When any DML is executed against a table with constraints defined, if the DML violates a constraint, then the whole statement will be rolled back automatically.

Remember that a DML statement that affects many rows might partially succeed before it hits a constraint problem with a particular row. If the statement is part of a multistatement transaction, then the statements that have already succeeded will remain intact but uncommitted.

The Types of Constraints

The constraint types supported by the Oracle database are as follows:

UNIQUE
NOT NULL
PRIMARY KEY
FOREIGN KEY
CHECK

Constraints have names. It is good practice to specify the names with a standard naming convention, but if they are not explicitly named, Oracle will generate names.

Unique Constraints

A unique constraint nominates a column (or combination of columns) for which the value must be different for every row in the table. If based on a single column, this is known as the *key* column. If the constraint is composed of more than one column (known as a *composite key* unique constraint) the columns do not have to be the same data type or be adjacent in the table definition.

An oddity of unique constraints is that it is possible to enter a NULL value into the key column(s); it is indeed possible to have any number of rows with NULL values in their key column(s). So selecting rows on a key column will guarantee that only one row is returned—unless you search for NULL, in which case all the rows where the key columns are NULL will be returned.

Unique constraints are enforced by an index. When a unique constraint is defined, Oracle will look for an index on the key column(s), and if one does not exist it will be created. Then whenever a row is inserted, Oracle will search the index to see if the values of the key columns are already present: if they are, it will reject the insert. The structure of these indexes (known as B*Tree indexes) does not include NULL values, which is why many rows with NULL are permitted: they simply do not exist in the index. While the first purpose of the index is to enforce the constraint, it has a secondary effect: improving performance if the key columns are used in the WHERE clauses of SQL statements. However, selecting `WHERE` *key_column* `IS NULL` cannot use the index because it doesn't include the NULLs and will therefore always result in a scan of the entire table.

Not Null Constraints

The not null constraint forces values to be entered into the key column. Not null constraints are defined per column: if the business requirement is that a group of columns should all have values, you cannot define one not null constraint for the whole group, but instead must define a not null constraint for each column.

Any attempt to insert a row without specifying values for the not null constrained columns results in an error. It is possible to bypass the need to specify a value by including a DEFAULT clause on the column when creating the table.

Primary Key Constraints

The primary key is the means of locating a single row in a table. The relational database paradigm includes a requirement that every table should have a primary key, a column (or combination of columns) that can be used to distinguish every row. The Oracle database deviates from the paradigm (as do some other RDBMS implementations) by permitting tables without primary keys.

The implementation of a primary key constraint is, in effect, the union of a unique constraint and a not null constraint. The key columns must have unique values, and they may not be null. As with unique constraints, an index must exist on the constrained column(s). If one does not exist already, an index will be created when the constraint is defined. A table can have only one primary key. Try to create a second, and you will get an error. A table can, however, have any number of unique constraints and not null columns, so if there are several columns that the business analysts have decided must be unique and populated, one of these can be designated the primary key and the others made unique and not null. An example could be a table of employees, where e-mail address, social security number, and employee number should all be required and unique.

Tables without primary keys are possible but not a good idea. Even if the business rules do not require the ability to identify every row, primary keys are often needed for maintenance work.

A primary key constraint is a unique constraint combined with a not null constraint.

Foreign Key Constraints

A foreign key constraint is defined on the child table in a parent-child relationship. The constraint nominates a column (or columns) in the child table that corresponds to the primary key column(s) in the parent table. The columns do not have to have the same names, but they must be of the same data type. Foreign key constraints define the relational structure of the database: the many-to-one relationships that connect the table, in their third normal form.

If the parent table has unique constraints as well as (or instead of) a primary key constraint, these columns can be used as the basis of foreign key constraints, even if they are nullable.

Just as a unique constraint permits null values in the constrained column, so does a foreign key constraint. You can insert rows into the child table with null foreign key columns—even if there is not a row in the parent table with a null value. This creates *orphan* rows and can cause dreadful confusion. As a general rule, all the columns in a unique constraint and all the columns in a foreign key constraint are best defined with not null constraints as well; this will often be a business requirement.

Attempting to inset a row in the child table for which there is no matching row in the parent table will give an error. Similarly, deleting a row in the parent table will give an error if there are already rows referring to it in the child table. There are two techniques for changing this behavior.

First, the constraint may be created as ON DELETE CASCADE. This means that if a row in the parent table is deleted, Oracle will search the child table for all the matching rows and delete them too. This will happen automatically.

A less drastic technique is to create the constraint as ON DELETE SET NULL. In this case, if a row in the parent table is deleted, Oracle will search the child table for all the matching rows and set the foreign key columns to null. This means that the child rows will be orphaned, but will still exist. If the columns in the child table also have a not null constraint, then the deletion from the parent table will fail.

It is not possible to drop or truncate the parent table in a foreign key relationship, even if there are no rows in the child table. This still applies if the ON DELETE SET NULL or ON DELETE CASCADE clauses were used.

A variation on the foreign key constraint is the *self-referencing* foreign key constraint. This defines a condition where the parent and child rows exist in the same table.

An example would be a table of employees, which includes a column for the employee's manager. The manager is himself an employee and must exist in the table. So if the primary key is the EMPLOYEE_NUMBER column, and the manager is identified by a column MANAGER_NUMBER, then the foreign key constraint will state that the value of the MANAGER_NUMBER column must refer back to a valid EMPLOYEE_NUMBER. If an employee is his own manager, then the row would refer to itself.

A foreign key constraint in a child table must reference the columns of either a unique constraint or a primary key constraint in the parent table.

Check Constraints

A check constraint can be used to enforce simple rules, such as that the value entered in a column must be within a range of values. The rule must be an expression which will evaluate to TRUE or FALSE. The rules can refer to absolute values entered as literals or to other columns in the same row and may make use of some functions.

As many check constraints as you want can be applied to one column, but it is not possible to use a subquery to evaluate whether a value is permissible or to use functions such as SYSDATE.

The not null constraint is in fact implemented as a preconfigured check constraint.

Defining Constraints

Constraints can be defined when creating a table or added to the table later. When defining constraints at table creation time, the constraint can be defined in line with the column to which it refers or at the end of the table definition. There is more flexibility to using the latter technique. For example, it is impossible to define a foreign key constraint that refers to two columns, or a check constraint that refers to any column other than that being constrained if the constraint is defined in line, but both of these are possible if the constraint is defined at the end of the table.

You are designing table structures for a human resources application. The business analysts have said that when an employee leaves the company, his employee record should be moved to an archive table. Can constraints help?

Probably not. Constraints are intended to enforce simple business rules: this may be too complicated. It may well be necessary to use a DML trigger on the live table, which will automatically insert a row into the archive table whenever an employee is deleted from the live table. Triggers can do much more complicated processing than a constraint.

Active transactions block some DDL statements against tables. If you want to add a constraint or rename a column in a busy table and find the statement always fails with "ORA-00054: resource busy and acquire with NOWAIT specified or timeout expired," what can you do?

Perhaps you shouldn't be doing this sort of thing when the database is in use, but should wait until the next period of scheduled downtime. However, if you really need to make the change in a hurry, ask the database administrator to quiesce the database: this is a process that will freeze all user sessions. If you are very quick, you can make the change then unquiesce the database before end users complain.

For the constraints that require an index (the unique and primary key constraints), the index will be created with the table if the constraint is defined at table creation time.

Consider these two table creation statements (to which line numbers have been added):

```
1 create table dept(
2 deptno number(2,0) constraint dept_deptno_pk primary key
3 constraint dept_deptno_ck check (deptno between 10 and 90),
4 dname varchar2(20) constraint dept_dname_nn not null);
5 create table emp(
6 empno number(4,0) constraint emp_empno_pk primary key,
7 ename varchar2(20) constraint emp_ename_nn not null,
8 mgr number (4,0) constraint emp_mgr_fk references emp (empno),
9 dob date,
10 hiredate date,
11 deptno number(2,0) constraint emp_deptno_fk references dept(deptno)
12 on delete set null,
13 email varchar2(30) constraint emp_email_uk unique,
14 constraint emp_hiredate_ck check (hiredate >= dob + 365*16),
15 constraint emp_email_ck
16 check ((instr(email,'@') > 0) and (instr(email,'.') > 0)));
```

Taking these statements line by line:

1. The first table created is DEPT, intended to have one row for each department.
2. DEPTNO is numeric, 2 digits, no decimals. This is the table's primary key. The constraint is named DEPT_DEPTNO_PK.
3. A second constraint applied to DEPTNO is a check limiting it to numbers in the range 10 to 90. The constraint is named DEPT_DEPTNO_CK.
4. The DNAME column is variable length characters, with a constraint DEPT_DNAME_NN making it not nullable.
5. The second table created is EMP, intended to have one row for every employee.
6. EMPNO is numeric, up to 4 digits with no decimals. Constraint EMP_EMPNO_PK marks this as the table's primary key.
7. ENAME is variable length characters, with a constraint EMP_ENAME_NN making it not nullable.
8. MGR is the employee's manager, who must himself be an employee. The column is defined in the same way as the table's primary key column of EMPNO. The constraint EMP_MGR_FK defines this column as a self-referencing foreign key, so any value entered must refer to an already extant row in EMP (though it is not constrained to be not null, so can be left blank).
9. DOB, the employee's birthday, is a date and not constrained.
10. HIREDATE is the date the employee was hired and is not constrained. At least, not yet.
11. DEPTNO is the department with which the employee is associated. The column is defined in the same way as the DEPT table's primary key column of DEPTNO, and the constraint EMP_DEPTNO_FK enforces a foreign key relationship: it is not possible to assign an employee to a department that does not exist. Though this is nullable.
12. The EMP_DEPTO_FK constraint is further defined as ON DELETE SET NULL, so if the parent row in DEPT is deleted, all matching child rows in EMPNO will have DEPTNO set to NULL.
13. EMAIL is variable length character data, and must be unique if entered (though it can be left empty).
14. This defines an additional table level constraint EMP_HIREDATE_CK. The constraint checks for use of child labor by rejecting any rows where the date of hiring is not at least 16 years later than the birthday. This constraint could not be defined in line with HIREDATE, because the syntax does not allow references to other columns at that point.
15. An additional constraint EMP_EMAIL_CK is added to the EMAIL column, which makes two checks on the e-mail address. The INSTR functions search for the at symbol (@ and dot (.) characters (which will always be present in a valid e-mail address); if it can't find both of them, the check condition will return FALSE and the row will be rejected.

The preceding examples show several possibilities for defining constraints at table creation time. The following are further possibilities not covered:

- Controlling the index creation for the unique and primary key constraints
- Defining whether the constraint should be checked at insert time (which it is by default) or later on when the transaction is committed
- Stating whether the constraint is in fact being enforced at all (which is the default) or is disabled.

It is possible to create tables with no constraints and then to add them later with an ALTER TABLE command. The end result will be the same, but this technique does make the code less self documenting, as the complete table definition will then be spread over several statements rather than being in one.

Work with Constraints

Use SQL*Plus or SQL Developer to create tables, add constraints, and demonstrate their use.

1. Connect to the database as user HR.
2. Create a table EMP as a copy of some columns from EMPLOYEES:

```
create table emp as
select employee_id empno, last_name ename, department_id deptno
from employees;
```

3. Create a table DEPT as a copy of some columns from DEPARTMENTS:

```
create table dept as
select department_id deptno, department_name dname from departments;
```

4. Use DESCRIBE to describe the structure of the new tables. Note that the not null constraint on ENAME and DNAME has been carried over from the source tables.
5. Add a primary key constraint to EMP and to DEPT and a foreign key constraint linking the tables:

```
alter table emp add constraint emp_pk primary key (empno);
alter table dept add constraint dept_pk primary key (deptno);
alter table emp add constraint
dept_fk foreign key (deptno) references dept on delete set null;
```

The preceding last constraint does not specify which column of DEPT to reference; this will default to the primary key column.

6. Demonstrate the effectiveness of the constraints by trying to insert data that will violate them:

```
insert into dept values(10,'New Department');
insert into emp values(9999,'New emp',99);
truncate table dept;
```

7. Tidy up by dropping the tables. Note that this must be done in the correct order:

```
drop table emp;
drop table dept;
```

TWO-MINUTE DRILL

Categorize the Main Database Objects

❑ Some objects contain data, principally tables and indexes.

❑ Programmatic objects such as stored procedures and functions are executable code.

❑ Views and synonyms are objects that give access to other objects.

Review the Table Structure

❑ Tables are two-dimensional structures, storing rows defined with columns.

❑ Tables exist within a schema. The schema name with the table name make a unique identifier.

List the Data Types that Are Available for Columns

❑ The most common character data types are VARCHAR2, NUMBER, and DATE.

❑ There are many other data types.

Create a Simple Table

❑ Tables can be created from nothing or with a subquery.

❑ After creation, column definitions can be added, dropped, or modified.

❑ The table definition can include default values for columns.

Explain How Constraints Are Created at the Time of Table Creation

❑ Constraints can be defined at table creation time or added later.

❑ A constraint can be defined inline with its column or at the table level after the columns.

❑ Table-level constraints can be more complex than those defined inline.

❑ A table may only have one primary key but can have many unique keys.

❑ A primary key is functionally equivalent to unique plus not null.

❑ A unique constraint does not stop insertion of many null values.

❑ Foreign key constraints define the relationships between tables.

SELF TEST

Choose all the correct answers for each question.

Categorize the Main Database Objects

1. If a table is created without specifying a schema, in which schema will it be? (Choose the best answer.)
A. It will be an *orphaned* table, without a schema.
B. The creation will fail.
C. It will be in the SYS schema.
D. It will be in the schema of the user creating it.
E. It will be in the PUBLIC schema.

2. Several object types share the same namespace, and therefore cannot have the same name in the same schema. Which of the following object types is not in the same namespace as the others? (Choose the best answer.)
A. Index
B. PL/SQL stored procedure
C. Synonym
D. Table
E. View

3. Which of these statements will fail because the table name is not legal? (Choose two answers.)
A. create table “SELECT” (col1 date);
B. create table “lowercase” (col1 date);
C. create table number1 (col1 date);
D. create table 1number (col1 date);
E. create table update (col1 date);

Review the Table Structure

4. What are distinguishing characteristics of heap tables? (Choose two answers.)
A. A heap can store variable length rows.
B. More than one table can store rows in a single heap.
C. Rows in a heap are in random order.
D. Heap tables cannot be indexed.
E. Tables in a heap do not have a primary key.

List the Data Types that Are Available for Columns

5. Which of the following data types are variable length? (Choose all correct answers.)
A. BLOB
B. CHAR
C. LONG
D. NUMBER
E. RAW
F. VARCHAR2

6. Study these statements:

```
create table tab1 (c1 number(1), c2 date);
alter session set nls_date_format='dd-mm-yy';
insert into tab1 values (1.1,’31-01-07’);
```

Will the insert succeed? (Choose the best answer)
A. The insert will fail because the 1.1 is too long.
B. The insert will fail because the ’31-01-07’ is a string, not a date.
C. The insert will fail for both reasons A and B.
D. The insert will succeed.

7. Which of the following is not supported by Oracle as an internal data type? (Choose the best answer.)
A. CHAR
B. FLOAT
C. INTEGER

D. STRING

Create a Simple Table

8. Consider this statement:

```
create table t1 as select * from regions where 1=2;
```

What will be the result? (Choose the best answer.)

A. There will be an error because of the impossible condition.
B. No table will be created because the condition returns FALSE.
C. The table T1 will be created but no rows inserted because the condition returns FALSE.
D. The table T1 will be created and every row in REGIONS inserted because the condition returns a NULL as a row filter.

9. When a table is created with a statement such as the following:

```
create table newtab as select * from tab;
```

will there be any constraints on the new table? (Choose the best answer.)

A. The new table will have no constraints, because constraints are not copied when creating tables with a subquery.
B. All the constraints on TAB will be copied to NEWTAB.
C. Primary key and unique constraints will be copied but not check and not null constraints.
D. Check and not null constraints will be copied but not unique or primary key.
E. All constraints will be copied, except foreign key constraints.

Explain How Constraints Are Created at the Time of Table Creation

10. Which types of constraint require an index? (Choose all that apply.)

A. CHECK
B. NOT NULL
C. PRIMARY KEY
D. UNIQUE

11. A transaction consists of two statements. The first succeeds, but the second (which updates several rows) fails partway through because of a constraint violation. What will happen? (Choose the best answer.)

A. The whole transaction will be rolled back.
B. The second statement will be rolled back completely, and the first will be committed.
C. The second statement will be rolled back completely, and the first will remain uncommitted.
D. Only the one update that caused the violation will be rolled back; everything else will be committed.
E. Only the one update that caused the violation will be rolled back; everything else will remain uncommitted.

LAB QUESTION

Consider this simple analysis of a telephone billing system: A subscriber is identified by a customer number and also has a name and possibly one or more telephones. A telephone is identified by its number, which must be a 7-digit integer beginning with 2 or 3, and also has a make, an activation date, and a flag for whether it is active. Inactive telephones are not assigned to a subscriber; active telephones are.

For every call, it is necessary to record the time it started and the time it finished. Create tables with constraints and defaults that can be used to implement this system.

SELF TEST ANSWERS

Categorize the Main Database Objects

1. ✹✓ D. The schema will default to the current user.

✹Y **A, B, C, E. A** is wrong because all tables must be in a schema. **B** is wrong because the creation will succeed. **C** is wrong because the SYS schema is not a default schema. **E** is wrong because while there is a notional user PUBLIC, he does not have a schema at all.

2. ✹✓ A. Indexes have their own namespace.

✹Y **B, C, D, E.** Stored procedures, synonyms, tables, and views exist in the same namespace.

3. ✹✓ D, E. D violates the rule that a table name must begin with a letter, and **E** violates the rule that a table name cannot be a reserved word. Both rules can be bypassed by using double quotes.

✹Y **A, B, C.** These are wrong because all will succeed (though **A** and **B** are not exactly sensible).

Review the Table Structure

4. ✹✓ A, C. A heap is a table of variable length rows in random order.

✹Y **B, D, E. B** is wrong because a heap table can only be one table. **D** and **E** are wrong because a heap table can (and usually will) have indexes and a primary key.

List the Data Types that Are Available for Columns

5. ✹✓ A, C, D, E, F. All these are variable length data types.

✹Y **B.** CHAR columns are fixed length.

6. ✹✓ D. The number will be rounded to 1 digit, and the string will cast as a date.

✹Y **A, B, C.** Automatic rounding and type casting will correct the “errors,” though ideally they would not occur.

7. ✹✓ D. STRING is not an internal data type.

✹Y **A, B, C.** CHAR, FLOAT, and INTEGER are all internal data types, though not as widely used as some others.

Create a Simple Table

8. ✹✓ C. The condition applies only to the rows selected for insert, not to the table creation.

✹Y **A, B, D. A** is wrong because the statement is syntactically correct. **B** is wrong because the condition does not apply to the DDL, only to the DML. **D** is wrong because the condition will exclude all rows from selection.

9. ✹✓ D. Check and not null constraints are not dependent on any structures other than the table to which they apply and so can safely be copied to a new table.

✹Y **A, B,C, E. A** is wrong because not null and check constraint will be applied to the new table. **B**, **C**, and **E** are wrong because these constraints need other objects (indexes or a parent table) and so are not copied.

Explain How Constraints Are Created at the Time of Table Creation

10. ✹✓ C, D. Unique and primary key constraints are enforced with indexes.

✹Y **A, B.** Check and not null constraints do not rely on indexes.

11. ✹✓ C. A constraint violation will force a roll back of the current statement but nothing else.

✹Y **A, B, D, E. A** is wrong because all statements that have succeeded remain intact. **B** and **D** are wrong because there is no commit of anything until it is specifically requested. **E** is wrong because the whole statement will be rolled back, not just the failed row.

LAB ANSWER

A possible solution:

- The SUBSCRIBERS table:

```
create table subscribers (id number(4,0) constraint sub_id_pk primary key,
name varchar2(20) constraint sub_name_nn not null);
```

- The TELEPHONES table:

```
create table telephones
(telno number (7,0) constraint tel_telno_pk primary key
constraint tel_telno_ck check (telno between 1000000 and 9999999),
activated date default sysdate,
active varchar2(1) constraint tel_active_nn not null
constraint tel_active_ck check(active='Y' or active='N'),
subscriber number(4,0) constraint tel_sub_fk references subscribers,
constraint tel_active_yn check((active='Y' and subscriber is not null)
or (active='N' and subscriber is null))
);
```

This table has a constraint on the ACTIVE column that will permit only Y or N, depending on whether there is a value in the SUBSCRIBER column. This constraint is too complex to define in line with the column, because it references other columns. SUBSCRIBER, if not null, must match a row in SUBSCRIBERS.

- The CALLS table:

```
create table calls
(telno number (7,0) constraint calls_telno_fk references telephones,
starttime date constraint calls_start_nn not null,
endtime date constraint calls_end_nn not null,
constraint calls_pk primary key(telno,starttime),
constraint calls_endtime_ck check(endtime > starttime));
```

Two constraints are defined at the table level, not the column level. The primary key cannot be defined in line with a column because it is based on two columns: one telephone can make many calls, but not two that begin at the same time (at least, not with current technology). The final constraint compares the start and end times of the call and so cannot be defined in line.

Chapter 11: Creating Other Schema Objects

In practice, most users never issue SQL that addresses tables: they issue SQL that addresses views or synonyms. Views and synonyms do not themselves store data; they provide a layer of abstraction between the users and the data.

As tables, views, and synonyms share the same namespace, users need never be aware of which they are addressing.
Sequences are a mechanism for issuing unique numbers. In many databases, primary key values are defined as a unique number. A sequence can issue such numbers on demand, without programmers needing to worry about whether they really are unique. The structure of a sequence means that hundreds of numbers can, if necessary, be issued every second without any performance issues.
Index management is sometimes said to lie in the database administration domain, not the SQL developer's domain. For this reason, the treatment here is very brief. But whether or not index management is part of the developer's job, all developers must understand the purpose and design of indexes. In many cases the indexing strategy is crucial to adequate performance: get it right, and the benefits will be enormous; get it wrong, and the results may be disastrous. Use of indexes is transparent, but if the programmers are aware of them they can ensure that their code will be able to make the best use of what indexes are available and advise on what indexes should be created.

Create Simple and Complex Views

To the user, a view looks like a table: a two-dimensional structure of rows of columns, against which the user can run SELECT and DML statements. The programmer knows the truth: a view is just a SELECT statement. Any SELECT statement returns a two-dimensional set of rows. If the SELECT statement is saved as a view, then whenever the users query or update rows in the view (under the impression that it is table), the statement runs, and the result is presented to users as though it were a table. The SELECT statement on which a view is based can be anything. It can join tables, perform aggregations, or do sorts; absolutely anything that is legal in the SELECT command can be used. However, if the view is *complex*, then only SELECT statements can be run against it.
A *simple* view is one which can be addressed by DML statements as well as SELECT. As might be expected, simple views are based on relatively simple SELECT statements; complex views are based on more complicated statements.

Why Use Views at All?

It is usually not a good idea to let end users loose on tables. Possible reasons include:

- Security.
- Simplifying user SQL.
- Preventing error.
- Making data comprehensible. Table and column names are often long and pretty meaningless. The view and its columns can be much more obvious.
- Performance.

Examples of views for each of these purposes follow.

Views to Enforce Security

It may be that users should only see certain rows or columns of a table. Consider the HR.EMPLOYEES table. This includes personal details that should not be visible to staff outside the personnel department. But finance staff will need to be able to see the costing information. This view will depersonalize the data:

```
create view hr.emp_fin as select
hire_date,job_id,salary,commission_pct,department_id from hr.employees;
```

Note the use of schema qualifiers for the table as the source of the data (often referred to as either the *base* or the *detail* table) and the view: views are schema objects and can draw their data from tables in the same schema or in other schemas. If the schema is not specified, it will of course be in the current schema.

Finance staff can then be given permission to see the view but not the table and can issue statements such as this:

```
select * from emp_fin where department_id=50;
```

They will see only the five columns that make up the view, not the remaining columns of EMPLOYEES with the personal information.
The view can be joined to other tables or aggregated as though it were a table:

```
select department_name, sum(salary) from departments natural join emp_fin
group by department_name;
```

A well constructed set of views can implement a whole security structure within the database, giving users access to data they need to see while concealing data they do not need to see.

Views to Simplify User SQL

It will be much easier for users to query data if the hard work (such as joins or aggregations) is done for them by the code that defines the view.
In the last example, the user had to write code that joined the EMP_FIN view to the DEPARTMENTS table and summed the salaries per department. This could all be done in a view:

```
create view dept_sal as
select d.department_name, sum(e.salary) from
departments d left outer join employees e on d.department_
id=e.department_id
group by department_name order by department_name;
```

Then the users can select from DEPT_SAL without needing to know anything about joins, or even how to sort the results:

```
select * from dept_sal;
```

In particular, they do not need to know how to make sure that all departments are listed, even those with no employees. The example in the previous section would have missed these.

Views to Prevent Errors

It is impossible to prevent users making errors, but well-constructed views can prevent some errors arising from a lack of understanding of how data should be interpreted. The previous section already introduced this concept by constructing a view that will list all departments, irrespective of whether they currently have staff assigned to them.
A view can help to present data in a way that is unambiguous. For example, many aplications never actually delete rows. Consider this table:

```
create table emp(empno number constraint emp_empno_pk primary key,
ename varchar2(10),deptno number,active varchar2(1) default 'Y');
```

The column ACTIVE is a flag indicating that the employee is currently employed and will default to 'Y' when a row is inserted. When a user, through the user interface, "deletes" an employee, the underlying SQL statement will be an update that sets ACTIVE to 'N'. If users who are not aware of this query the table, they may severely misinterpret the results. It will often be better to give them access to a view:

```
create view current_staff as select * from emp where active='Y';
```

Queries addressed to this view cannot possibly see "deleted" staff members.

Views to Make Data Comprehensible

The data structures in a database will be normalized tables. It is not reasonable to expect users to understand normalized structures.
To take an example from the Oracle E-Business suite, a "customer" in the Accounts Receivable module is in fact an entity consisting

of information distributed across the tables HZ_PARTIES, HZ_PARTY_SITES, HZ_CUST_ACCTS_ALL, and many more. All these tables are linked by primary key–to–foreign key relationships, but these are not defined on any identifiers visible to users (such as a customer number): they are based on columns the users never see that have values generated internally from sequences.

The forms and reports used to retrieve customer information never address these tables directly; they all work through views. As well as presenting data to users in a comprehensible form, the use of views to provide a layer of abstraction between the objects seen by users and the objects stored within the database can be invaluable for maintenance work. It becomes possible to redesign the data structures without having to recode the application. If tables are changed, then adjusting the view definitions may make any changes to the SQL and PL/SQL code unnecessary. This can be a powerful technique for making applications portable across different databases.

Views for Performance

The SELECT statement behind a view can be optimized by programmers, so that users don't need to worry about tuning code. There may be many possibilities for getting the same result, but some techniques can be much slower than others. For example, when joining two tables there is usually a choice between the *nested loop* join and the *hash* join. A nested loop join uses an index to get to individual rows; a hash join reads the whole table into memory. The choice between the two will be dependent on the state of the data and the hardware resources available. Theoretically, one can always rely on the Oracle optimizer to work out the best way to run a SQL statement, but there are cases where it gets it wrong. If the programmers know which technique is best, they can instruct the optimizer. This example forces use of the hash technique:

```
create view dept_emp as
select /*+USE_HASH (employees departments)*/ department_name, last_name
from departments natural join employees;
```

Whenever users query the DEPT_EMP view, the join will be performed by scanning the detail tables into memory. The users need not know the syntax for forcing use of this join method.

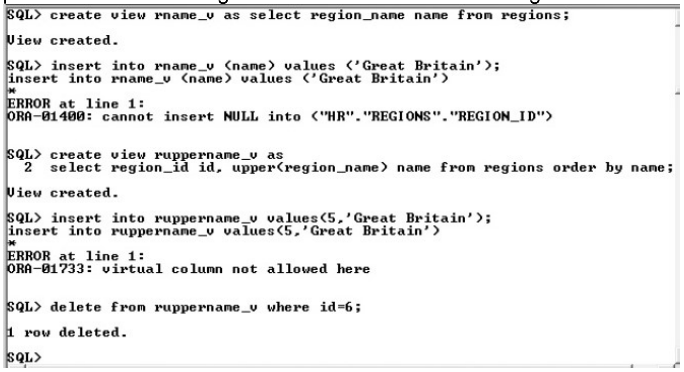
Simple and Complex Views

For practical purposes, classification of a view as *simple* or *complex* is related to whether DML statements can be executed against it: simple views can accept DML statements, complex views can't. The strict definitions are as follows:

- A simple view draws data from one detail table, uses no functions, and does no aggregation.
- A complex view can join detail tables, use functions, and perform aggregations.

Applying these definitions shows that of the four views used as examples in the previous section, the first and third are simple and the second and fourth are complex.

It is often not possible to execute INSERT, UPDATE, or DELETE commands against a complex view. The mapping of the rows in the view back to the rows in the detail table(s) cannot always be established on a one-to-one basis, which is necessary for DML operations. It is usually possible to execute DML against a simple view but not always. For example, if the view does not include a column that has a NOT NULL constraint, then an INSERT through the view cannot succeed (unless the column has a default value). This can produce a disconcerting effect because the error message will refer to a table and a column that are not mentioned in the statement.



```
SQL> create view rname_v as select region_name name from regions;

View created.

SQL> insert into rname_v (name) values ('Great Britain');
insert into rname_v (name) values ('Great Britain')
*
ERROR at line 1:
ORA-01400: cannot insert NULL into ("HR"."REGIONS"."REGION_ID")


SQL> create view ruppername_v as
  2  select region_id id, upper(region_name) name from regions order by name;

View created.

SQL> insert into ruppername_v values(5,'Great Britain');
insert into ruppername_v values(5,'Great Britain')
*
ERROR at line 1:
ORA-01733: virtual column not allowed here


SQL> delete from ruppername_v where id=6;

1 row deleted.

SQL>
```

The first view in the figure, RNAME_V, does conform to the definition of a simple view, but an INSERT cannot be performed through it because it is missing a mandatory column. The second view, RUPPERNAME_V, is a complex view because it includes a function. This makes an INSERT impossible, because there is no way the database can work out what should actually be inserted: it can't reverse engineer the effect of the UPPER function in a deterministic fashion. But the DELETE succeeds, because that is not dependent on the function.

As a rule, simple views accept DML, complex views don't. But there are exceptions.

CREATE VIEW, ALTER VIEW, and DROP VIEW

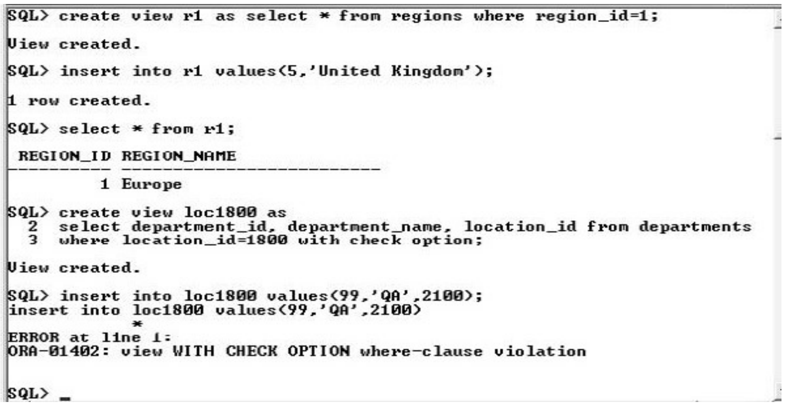
The syntax to create a view is as follows:

CREATE [OR REPLACE] [FORCE | NOFORCE] VIEW
[*schema*.]*viewname* [(*alias* [,*alias*]…)]
AS *subquery*
[WITH CHECK OPTION [CONSTRAINT *constraintnam*e]]
[WITH READ ONLY [CONSTRAINT *constraintname*]] ;

By default, the view will be created in the current schema. The optional keywords, none of which have been used in the examples so far, are as follows:

- **OR REPLACE** If the view already exists, it will be dropped before being created.
- **FORCE or NOFORCE** The FORCE keyword will create the view even if the detail table(s) in the subquery does not exist. NOFORCE is the default and will cause an error if the detail table does not exist.
- **WITH CHECK OPTION** This is to do with DML. If the subquery includes a WHERE clause, then this option will prevent insertion of rows that wouldn't be seen in the view or updates that would cause a row to disappear from the view. By default, this option is not enabled, which can give disconcerting results.
- **WITH READ ONLY** Prevents any DML through the view.
- **CONSTRAINT *constraintname*** This can be used to name the WITH CHECK OPTION and WITH READ ONLY restrictions so that error messages when the restrictions cause statements to fail, will be more comprehensible.

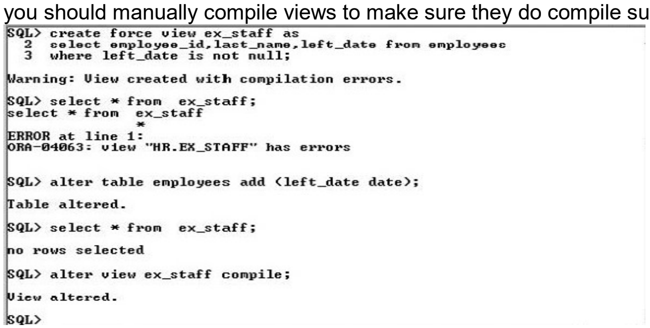
In addition, a set of alias names can be provided for the names of the view's columns. If not provided, the columns will be named after the table's columns or with aliases specified in the subquery.



```
SQL> create view r1 as select * from regions where region_id=1;

View created.

SQL> insert into r1 values(5,'United Kingdom');

1 row created.

SQL> select * from r1;

 REGION_ID REGION_NAME
---------- -------------------------
         1 Europe

SQL> create view loc1800 as
  2  select department_id, department_name, location_id from departments
  3  where location_id=1800 with check option;

View created.

SQL> insert into loc1800 values(99,'QA',2100);
insert into loc1800 values(99,'QA',2100)
            *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation


SQL> _
```

Creating a view, R1, which permits insertion of rows that cannot be seen. The second view, LOC1800, uses WITH CHECK OPTION to prevent this from happening.

The main use of the ALTER VIEW command is to compile the view. A view must be compiled successfully before it can be used. When a view is created, Oracle will check that the detail tables and the necessary columns on which the view is based do exist. If they do not, the compilation fails and the view will not be created—unless you use the FORCE option. In that case, the view will be created but will be unusable until the tables or columns to which it refers are created and the view is successfully compiled. When an invalid view is queried, Oracle will attempt to compile it automatically. If the compilation succeeds because the problem has been fixed, the user won’t know there was ever a problem—except that his query may take a little longer than usual. Generally speaking, you should manually compile views to make sure they do compile successfully, rather than having users discover errors.



```
SQL> create force view ex_staff as
  2  select employee_id,last_name,left_date from employees
  3  where left_date is not null;

Warning: View created with compilation errors.

SQL> select * from  ex_staff;
select * from  ex_staff
               *
ERROR at line 1:
ORA-04063: view "HR.EX_STAFF" has errors


SQL> alter table employees add (left_date date);

Table altered.

SQL> select * from  ex_staff;

no rows selected

SQL> alter view ex_staff compile;

View altered.

SQL>
```

In the figure, the first statement creates a view EX_STAFF that references a column EMPLOYEES.LEFT_DATE, which does not exist. The statement does not fail because the creation is forced, but it does give a warning. An attempt to select from the view fails. After adding the necessary column to the detail table, the SELECT succeeds. This is because the view was compiled successfully automatically.
Finally, there is a manual compilation of the view. It would have been better practice to do this immediately after adding the missing column.
It is not possible to adjust a view’s column definitions after creation in the way that a table’s columns can be changed. The view must be dropped and re-created.

The DROP command is as follows:
DROP VIEW [*schema*.]*viewname* ;
By using the OR REPLACE keywords with the CREATE VIEW command, the view will be automatically dropped (if it exists at all) before being created.

Create Views
In this exercise, you will create some simple and complex views, using data in the HR schema. Either SQL*Plus or SQL developer can be used.
1. Connect to your database as user HR.
2. Create views on the EMPLOYEES and DEPARTMENT tables that remove all personal information:

```
create view emp_anon_v as
select hire_date, job_id,salary,commission_pct,department_id from
employees;
create view dept_anon_v as
select department_id,department_name,location_id from departments;
```

3. Create a complex view that will join and aggregate the two simple views. Note that there is no reason not to have views of views.

```
create view dep_sum_v as
select e.department_id,count(1) staff, sum(e.salary) salaries,
d.department_name from emp_anon_v e join dept_anon_v d
on e.department_id=d.department_id
group by e.department_id,d.department_name;
```

4. Confirm that the view works by querying it.

Retrieve Data from Views
Queries can be addressed to views exactly as though to tables. Views and tables share the same namespace, so syntactically there is absolutely no difference in the SELECT statements. The user need have no knowledge of which type of object he is addressing.
Consider this view:

```
create view dept_emp as
select department_name, last_name from
departments join employees using (department_id);
```

The following query:

```
select * from dept_emp where department_name='Marketing';
```

when parsed by Oracle for execution will become:

```
select * from
(select department_name, last_name
from departments join employees using (department_id))
```

```
where department_name='Marketing';
```

You rarely get anything for nothing, and use of views certainly does not in any way reduce the work that the database has to do. Views can, however, reduce substantially the work that users have to do.

Use Views

In this exercise, you will use the views created previously for SELECT and DML statements. Either SQL*Plus or SQL developer can be used.

1. Connect to your database as user HR.
2. Insert a new department through the DEPT_ANON_V view and attempt to insert an employee through EMP_ANON_V:

```
insert into DEPT_ANON_V values(99,'Temp Dept',1800);
insert into emp_anon_v values(sysdate,'AC_MGR',10000,0,99);
```

The insert into EMP_ANON_V will fail because of missing mandatory values. You will, however, be able to do this update through it:

```
update emp_anon_v set salary=salary*1.1;
```

Then roll back the changes:

```
rollback;
```

3. Find out the average salary of the department with the highest average salary, by querying the EMPLOYEES table:

```
select max(avg_sal) from
(select AVG(SALARY) avg_sal from employees group by department_id);
```

and find the same information from the DEP_SUM_V view, which is a much simpler query:

```
select max(salaries / staff) from dep_sum_v;
```

Create Private and Public Synonyms

A synonym is an alternative name for an object. If synonyms exist for objects, then any SQL statement can address the object either by its actual name, or by its synonym. This may seem trivial. It isn't. Use of synonyms means that an application can function for any user, irrespective of which schema owns the views and tables or even in which database the tables reside. Consider this statement:

```
select * from hr.employees@prod;
```

The user issuing the statement must know that the employees table is owned by the HR schema in the database identified by the database link PROD (do not worry about database links—they are a means of accessing objects in a database other than that onto which you are logged). If a public synonym has been created with this statement:

```
create public synonym emp for hr.employees@prod;
```

then all the user (any user!) need enter is the following:

```
select * from emp;
```

This gives both data independence and location transparency. Tables and views can be renamed or relocated without ever having to change code; only the synonyms need to be adjusted.

As well as SELECT statements, DML statements can address synonyms as though they were the object to which they refer.

Private synonyms are schema objects. Either they must be in your own schema, or they must be qualified with the schema name. Public synonyms exist independently of a schema. A public synonym can be referred to by any user to whom permission has been granted to see it without the need to qualify it with a schema name. Private synonyms must be a unique name within their schema. Public synonyms can have the same name as schema objects. When executing statements that address objects without a schema qualifier, Oracle will first look for the object in the local schema, and only if it cannot be found will it look for a public synonym. Thus, in the preceding example, if the user happened to own a table called EMP it would be this that he would see—not the table pointed to by the public synonym.

The syntax to create a synonym is as follows:

CREATE [PUBLIC] SYNONYM *synonym* FOR *object* ;

A user will need to have been granted permission to create private synonyms and further permission to create public synonyms. Usually, only the database administrator can create (or drop) public synonyms. This is because their presence (or absence) will affect every user.

To drop a synonym:

DROP [PUBLIC] SYNONYM *synonym* ;

If the object to which a synonym refers (the table or view) is dropped, the synonym continues to exist. Any attempt to use it will return an error. In this respect, synonyms behave in the same way as views. If the object is recreated, the synonym must be recompiled before use. As with views, this will happen automatically the next time the synonym is addressed, or it can be done explicitly with

ALTER SYNONYM *synonym* COMPILE;

The "public" in "public synonym" means that it is not a schema object and cannot therefore be prefixed with a schema name. It does not mean that everyone has permissions against it.

Create and Use Synonyms

In this exercise, you will create and use private synonyms, using objects in the HR schema. Either SQL*Plus or SQL developer can be used.

1. Connect to your database as user HR.
2. Create synonyms for the three views created in Exercise 11-1:

```
create synonym emp_s for emp_anon_v;
create synonym dept_s for dept_anon_v;
create synonym dsum_s for dep_sum_v;
```

3. Confirm that the synonyms are identical to the underlying object:

```
describe emp_s;
describe emp_anon_v;
```

4. Confirm that the synonyms work (even to the extent of producing the same errors) by running the statements in Exercises 11-1 and 11-2 against the synonyms instead of the views:

```
select * from dsum_s;
insert into dept_s values(99,'Temp Dept',1800);
insert into emp_s values(sysdate,'AC_MGR',10000,0,99);
update emp_s set salary=salary*1.1;
rollback;
select max(salaries / staff) from dsum_s;
```

5. Drop two of the views:

```
drop view emp_anon_v;
drop view dept_anon_v;
```

6. Query the complex view that is based on the dropped views:

```
select * from dep_sum_v;
```

Note that the query fails.

7. Attempt to recompile the broken view:

```
alter view dep_sum_v compile;
```

This will fail as well.

8. Drop the DEP_SUM_V view:

```
drop view dep_sum_v;
```

9. Query the synonym for a dropped view:

```
select * from emp_s;
```

This will fail.

10. Recompile the broken synonym:

```
alter synonym emp_s compile;
```

Note that this does not give an error, but rerun the query from step 9. It is definitely still broken.

11. Tidy up by dropping the synonyms:

```
drop synonym emp_s;
drop synonym dept_s;
drop synonym dsum_s;
```

Create, Maintain, and Use Sequences

A sequence is a structure for generating unique integer values. Only one session can read the next value and thus force it to increment. This is a point of serialization, so each value generated will be unique.

Sequences are an invaluable tool for generating primary keys. Many applications will need automatically generated primary key values.

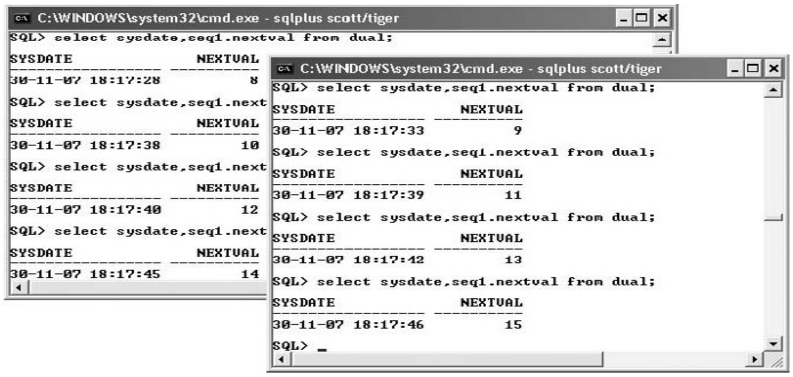
Examples in everyday business data processing are invoice numbers or order numbers: the business analysts will have stated that every invoice and order must have a unique number, which should continually increment. Other applications may not have such a requirement in business terms, but it will be needed to enforce relational integrity. Consider a telephone billing system: in business terms the unique identifier of a telephone is the telephone number (which is a string) and that of a call will be the source telephone number and the time the call began (which is a timestamp). These data types are unnecessarily complex to use as primary keys for the high volumes that go through a telephone switching system. While this information will be recorded, it will be much faster to use simple numeric columns to define the primary and foreign keys. The values in these columns can be sequence based.

The sequence mechanism is independent of tables, the row locking mechanism, and commit or rollback processing. This means that a sequence can issue thousands of unique values a minute—far faster than any method involving selecting a column from a table, updating it, and committing the change.

Creating Sequences

The full syntax for creating a sequence is as follows:

CREATE SEQUENCE [*schema*.]*sequencename*
[INCREMENT BY *number*]
[START WITH *number*]
[MAXVALUE *number* | NOMAXVALUE]
[MINVALUE *number* | NOMINVALUE]
[CYCLE | NOCYCLE]
[CACHE *number* | NOCACHE]
[ORDER | NOORDER] ;

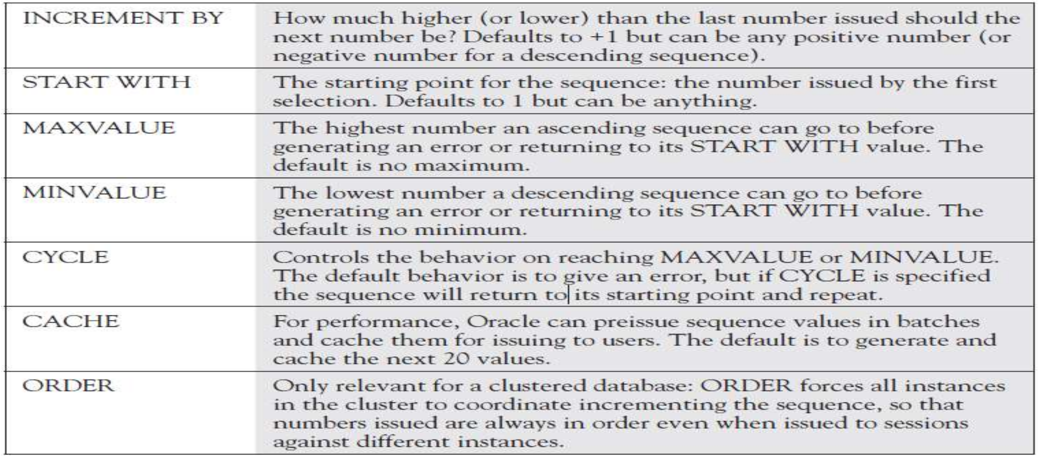

C:\WINDOWS\system32\cmd.exe - sqlplus scott/tiger
C:\WINDOWS\system32\cmd.exe - sqlplus scott/tiger

```
SQL> select sysdate,seq1.nextval from dual;

SYSDATE              NEXTVAL
------------------- ----------
30-11-07 18:17:28          8

SQL> select sysdate,seq1.next
SYSDATE              NEXTVAL
------------------- ----------
30-11-07 18:17:38         10

SQL> select sysdate,seq1.next
SYSDATE              NEXTVAL
------------------- ----------
30-11-07 18:17:40         12

SQL> select sysdate,seq1.next
SYSDATE              NEXTVAL
------------------- ----------
30-11-07 18:17:45         14
```

```
SQL> select sysdate,seq1.nextval from dual;
SYSDATE              NEXTVAL
------------------- ----------
30-11-07 18:17:33          9

SQL> select sysdate,seq1.nextval from dual;
SYSDATE              NEXTVAL
------------------- ----------
30-11-07 18:17:39         11

SQL> select sysdate,seq1.nextval from dual;
SYSDATE              NEXTVAL
------------------- ----------
30-11-07 18:17:42         13

SQL> select sysdate,seq1.nextval from dual;
SYSDATE              NEXTVAL
------------------- ----------
30-11-07 18:17:46         15

SQL> _
```

```
create sequence seq1;
```

The options are shown in the following table.



INCREMENT BY	How much higher (or lower) than the last number issued should the next number be? Defaults to +1 but can be any positive number (or negative number for a descending sequence).
START WITH	The starting point for the sequence: the number issued by the first selection. Defaults to 1 but can be anything.
MAXVALUE	The highest number an ascending sequence can go to before generating an error or returning to its START WITH value. The default is no maximum.
MINVALUE	The lowest number a descending sequence can go to before generating an error or returning to its START WITH value. The default is no minimum.
CYCLE	Controls the behavior on reaching MAXVALUE or MINVALUE. The default behavior is to give an error, but if CYCLE is specified the sequence will return to its starting point and repeat.
CACHE	For performance, Oracle can preissue sequence values in batches and cache them for issuing to users. The default is to generate and cache the next 20 values.
ORDER	Only relevant for a clustered database: ORDER forces all instances in the cluster to coordinate incrementing the sequence, so that numbers issued are always in order even when issued to sessions against different instances.

Appropriate settings for INCREMENT BY, START WITH, and MAXVALUE or MINVALUE will come from your business analysts.

It is very rare for CYCLE to be used because it lets the sequence issue duplicate values. If the sequence is being used to generate primary key values, CYCLE only makes sense if there is a routine in the database that will delete old rows faster than the sequence will reissue numbers.

Caching sequence values is vital for performance. Selecting from a sequence is a point of serialization in the application code: only one session can do this at once.

The mechanism is very efficient: it is much faster than locking a row, updating the row, and then unlocking it with a COMMIT. But even so, selecting from a sequence can be a cause of contention between sessions. The CACHE keyword instructs Oracle to pregenerate sequence numbers in batches. This means that they can be issued faster than if they had to be generated on demand.

The default number of values to cache is only 20. Experience shows that this is not enough. If your application selects from the sequence 10 times a second, then set the cache value to 50 thousand. Don't be shy about this.

Using Sequences

To use a sequence, a session can select either the next value with the NEXTVAL pseudo column, which forces the sequence to increment, or the last (or "current") value issued to that session with the CURRVAL pseudo column. The NEXTVAL will be globally unique: each session that selects it will get a different, incremented, value for each SELECT. The CURRVAL will be constant for one session until it selects NEXTVAL again. There is no way to find out what the last value issued by a sequence was: you can always obtain the next value by incrementing it with NEXTVAL, and you can always recall the last value issued to your session with CURRVAL, but you cannot find the last value issued.

A typical use of sequences is for primary key values. This example uses a sequence ORDER_SEQ to generate unique order numbers and LINE_SEQ to generate unique line numbers for the line items of the order. First create the sequences, which is a one-off operation:

```
create sequence order_seq start with 10;
create sequence line_seq start with 10;
```

Then insert the orders with their lines as a single transaction:

```
insert into orders (order_number,order_date,customer_number)
values (order_seq.nextval,sysdate,'1000');
insert into order_lines (order_number,line_number,item_number,quantity)
values (order_seq.currval,line_seq.nextval,'A111',1);
insert into order_lines (order_number,line_number,item_number,quantity)
values (order_seq.currval,line_seq.nextval,'B111',1);
commit;
```

The CURRVAL of a sequence is the last value issued to the current session, not necessarily the last value issued.
You cannot select the CURRVAL until after selecting the NEXTVAL.

The first INSERT statement raises an order with a unique order number drawn from the sequence ORDER_SEQ for customer number 1000. The second and third statements insert the two lines of the order, using the previously issued order number from ORDER_SEQ as the foreign key to connect the lines to the order, and the next values from LINE_SEQ to generate a unique identifier for each line. Finally, the transaction is committed.

A sequence is not tied to any one table. In the preceding example, there would be no technical reason not to use one sequence to generate values for the primary keys of the order and of the lines.

A COMMIT is not necessary to make the increment of a sequence permanent: it is permanent and made visible to the rest of the world the moment it happens. It can't be rolled back, either.

The second insert is rolled back. But as the final query shows, the sequence was incremented in spite of this. Sequence updates occur independently of the transaction management system. For this reason, there will always be gaps in the series. The gaps will be larger if the database has been restarted and the CACHE clause was used. All numbers that have been generated and cached but not yet issued will be lost when the database is shut down. At the next restart, the current value of the sequence will be the last number generated, not the last issued. So, with the default CACHE of 20, every shutdown/startup will lose up to 20 numbers.

If the business analysts have stated that there must be no gaps in a sequence, then another means of generating unique numbers must be used. For the preceding example of raising orders, the current order number could be stored in this table and initialized to 10:

```
create table current_on(order_number number);
insert into current_on values(10);
commit;
```

Then the code to create an order would have to become:

```
update current_on set order_number=order_number + 1;
insert into orders (order_number,order_date,customer_number)
values ((select order_number from current_on),sysdate,'1000');
commit;
```

This will certainly work as a means of generating unique order numbers, and because the increment of the order number is within the transaction that inserts the order, it can be rolled back with the insert if necessary: there will be no gaps in order numbers, unless an order is deliberately deleted. But it is far less efficient than using a sequence, and code like this is famous for causing dreadful contention problems. If many sessions try to lock and increment the one row containing the current number, the whole application will hang as they queue up to take their turn. After creating and using a sequence, it can be modified. The syntax is as follows:

ALTER SEQUENCE *sequencename*
[INCREMENT BY *number*]
[START WITH *number*]
[MAXVALUE *number* | NOMAXVALUE]
[MINVALUE *number* | NOMINVALUE]
[CYCLE | NOCYCLE]
[CACHE *number* | NOCACHE]
[ORDER | NOORDER] ;

This ALTER command is the same as the CREATE command, with one exception: there is no way to set the starting value. If you want to restart the sequence, the only way is to drop it and re-create it. To adjust the cache value from default to improve performance of the preceding order entry example:

```
alter sequence order_seq cache 1000;
```

However, if you want to reset the sequence to its starting value, the only way is to drop it:

```
drop sequence order_seq;
```

and create it again.

Create and Use Sequences

In this exercise, you will create some sequences and use them. You will need two concurrent sessions, either SQL Developer or SQL*Plus.

1. Log on to your database twice, as HR in separate sessions. Consider one to be your A session and the other to be your B session.

You are involved in designing a database to be used for online order entry and offline financial reporting. What should you consider with regard to views, synonyms, and indexes?

The data must be normalized for storage but should be denormalized into views for retrieval. The dual use of the database will be a problem: too many indexes will slow down the order entry, too few will impact adversely on reporting.

An alternative approach would be create a separate set of tables (possibly denormalized and aggregated like the views) for reporting and update these with batch routines. Using synonyms at all times would make it easier to switch between implementations with and without separate reporting tables. Should sequences always be used for primary keys?

In some cases, the business analysts may have defined a primary key based on attributes of the data. For example, a list of telephone subscribers could be keyed on telephone number: this is a formatted string that may have information such as geographical location (for a land line) or network operator (for a mobile) embedded within it. But usually a primary key does not have meaning, and in these cases a sequence is always the best way.

2. In your A session, create a sequence as follows:

```
create sequence seq1 start with 10 nocache maxvalue 15 cycle;
```

The use of NOCACHE is deleterious for performance. If MAXVALUE is specified, then CYCLE will be necessary to prevent errors when MAXVALUE is reached.

3. Execute the following commands in the appropriate session in the correct order to observe the use of NEXTVAL and CURRVAL and the cycling of the sequence:

1st select seq1.nextval from dual;
2nd select seq1.nextval from dual;
3rd select seq1.nextval from dual;
4th select seq1.nextval from dual;
5th select seq1.currval from dual;
6th select seq1.nextval from dual;
7th select seq1.nextval from dual;
8th select seq1.currval from dual;
9th select seq1.nextval from dual;
10th select seq1.nextval from dual;

4. Create a table with a primary key:

```
create table seqtest(c1 number,c2 varchar2(10));
alter table seqtest add constraint seqtest_pk primary key (c1);
```

5. Create a sequence to generate primary key values:

```
create sequence seqtest_pk_s;
```

6. In your A session, insert a row into the new table and commit:

```
insert into seqtest values(seqtest_pk_s.nextval,'first');
commit;
```

7. In your B session, insert a row into the new table and do not commit it:

```
insert into seqtest values(seqtest_pk_s.nextval,'second');
```

8. In your A session, insert a third row and commit:

```
insert into seqtest values(seqtest_pk_s.nextval,'third');
commit;
```

9. In your B session, roll back the second insertion:

```
rollback;
```

10. In your B session, see the contents of the table:

```
select * from seqtest;
```

This demonstrates that sequences are incremented and the next value published immediately, outside the transaction control mechanism.

11. Tidy up:

```
drop table seqtest;
drop sequence seqtest_pk_s;
drop sequence seq1;
```

Create and Maintain Indexes

Indexes have two functions: to enforce primary key and unique constraints and to improve performance. An application's indexing strategy is critical for performance. Theoretically, the SQL developer does not need to be aware of the existence (or otherwise) of indexes, but if the developer does know what indexes exist and how they are structured, the developer can write code designed to exploit them.

There is no clear demarcation of whose domain index management lies within. When the business analysts specify business rules that will be implemented as constraints, they are in effect specifying indexes. The database administrators will be monitoring the execution of code running in the database and will make recommendations for indexes. The developer, who should have the best idea of what is going on in the code and the nature of the data, will also be involved in developing the indexing strategy.

What Indexes Are For

Indexes are part of the constraint mechanism. If a column (or a group of columns) is marked as a table's primary key, then every time a row is inserted into the table, Oracle must check that a row with the same value in the primary key does not already exist. If the table has no index on the column(s), the only way to do this is to scan right through the table, checking every row. While this might be acceptable for a table of only a few rows, for a table with thousands or millions (or billions) of rows, this is not feasible. An index gives (near) immediate access to key values, so the check for existence can be made virtually instantaneously. When a primary key constraint is defined, Oracle will automatically create an index on the primary key column(s), if one does not exist already.

A unique constraint also requires an index. The difference from a primary key constraint is that the column(s) of the unique constraint can be left null, perhaps in many rows. This does not affect the creation and use of the index: nulls do not go into the B*Tree indexes, as described in the next section, "Types of Index."

Foreign key constraints are enforced by indexes, but the index must exist on the parent table, not necessarily on the table for which the constraint is defined. A foreign key constraint relates a column in the child table to the primary key or to a unique key in the parent table. When a row is inserted in the child table, Oracle will do a lookup on the index on the parent table to confirm that there is a matching row before permitting the insert. However, you should always create indexes on the foreign key columns within the child table for performance reasons: a DELETE on the parent table will be much faster if Oracle can use an index to determine whether there are any rows in the child table referencing the row that is being deleted.

Indexes are critical for performance. When executing any SQL statement that includes a WHERE clause, Oracle has to identify which rows of the table are to be selected or modified. If there is no index on the column(s) referenced in the WHERE clause, the only way to do this is with a *full table scan*. A full table scan reads every row of the table, in order to find the relevant rows. If the table has billions of rows, this can take hours. If there is an index on the relevant column(s), Oracle can search the index instead. An index is a sorted list of key values, structured in a manner that makes the search very efficient. With each key value is a pointer to the row in the table. Locating relevant rows via an index lookup is far faster than using a full table scan, if the table is over a certain size and the proportion of the rows to be retrieved is below a certain value. For small tables, or for a WHERE clause that will retrieve a large fraction of the table's rows, a full table scan will be quicker: you can (usually) trust Oracle to make the correct decision, based on information the database gathers about the tables and the rows within them.

A second circumstance where indexes can be used is for sorting. A SELECT statement that includes the ORDER BY, GROUP BY, or UNION keywords (and a few others) must sort the rows into order—unless there is an index, which can return the rows in the correct order without needing to sort them first.

A third circumstance when indexes can improve performance is when tables are joined, but again Oracle has a choice: depending on the size of the tables and the memory resources available, it may be quicker to scan tables into memory and join them there, rather than use indexes. The *nested loop join* technique passes through one table using an index on the other table to locate the matching rows: this is usually a disk-intensive operation. A *hash join* technique reads the entire table into memory, converts it into a hash table, and uses a hashing algorithm to locate matching rows; this is more memory and CPU intensive. A *sort merge join* sorts the tables on the join column then merges them together: this is often a compromise between disk, memory, and CPU resources. If there are no indexes, then Oracle is severely limited in the join techniques available.

Types of Index

Oracle supports several types of index, which have several variations. The two index types of concern here are the B*Tree index, which is the default index type, and the bitmap index. The only variation worthy of consideration for examination purposes applies to B*Tree indexes: these can be either *unique* or *nonunique*.

A unique index will not permit insertion of two rows with the same key values; a nonunique index will permit as many rows as you want with the same values. Nonunique is the default.

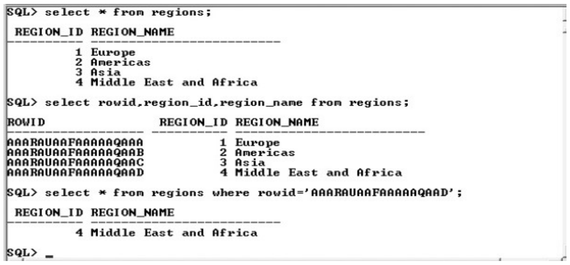
As a general rule, indexes will improve performance for data retrieval but reduce performance for DML operations. This is because indexes must be maintained. Every time a row is inserted into a table, a new key must be inserted into every index on the table, which places an additional strain on the database. For this reason, on transaction processing systems it is customary to keep the number of indexes as low as possible (perhaps no more than those needed for the constraints) and on a query intensive system, such as a data warehouse, to create as many as might be helpful.

B*Tree Indexes

A B*Tree index (the "B" stands for "balanced") is a tree structure. The root node of the tree points to many nodes at the second level, which can point to many nodes at the third level, and so on. The necessary depth of the tree will be largely determined by the number of rows in the table and the length of the index key values.

The leaf nodes of the index tree store the rows' keys, in order, each with a pointer that identifies the physical location of the row. So to retrieve a row with an index lookup, if the WHERE clause is using an equality predicate on the primary key column, Oracle navigates down the tree to the leaf node containing the desired key value, then uses the pointer to find the row. If the WHERE clause is using a nonequality predicate (such as any of the operators LIKE, BETWEEN, >, or <) then Oracle can navigate down the tree to find the first matching key value and then navigate across the leaf nodes of the index to find all the other matching values. As it does so, it will retrieve the rows from the table, in order.

The pointer to the row is the *rowid*. The rowid is an Oracle proprietary pseudocolumn that every row in every table has. Encrypted within it is the physical address of the row. As rowids are not part of the SQL standard, they are never visible to a normal SQL statement, but you can see them and use them if you want.



```
SQL> select * from regions;

 REGION_ID REGION_NAME
---------- -------------------------
         1 Europe
         2 Americas
         3 Asia
         4 Middle East and Africa

SQL> select rowid,region_id,region_name from regions;

ROWID               REGION_ID REGION_NAME
------------------ ---------- -------------------------
AAARAUAAFAAAAAQAAA          1 Europe
AAARAUAAFAAAAAQAAB          2 Americas
AAARAUAAFAAAAAQAAC          3 Asia
AAARAUAAFAAAAAQAAD          4 Middle East and Africa

SQL> select * from regions where rowid='AAARAUAAFAAAAAQAAD';

 REGION_ID REGION_NAME
---------- -------------------------
         4 Middle East and Africa

SQL> _
```

A row's rowid is globally unique. Every row in every table in the whole database will have a different rowid. The rowid encryption gives the physical address of the row: from it, Oracle can calculate which operating system file and where in the file the row is, and go straight to it.

B*Tree indexes are a very efficient way of retrieving rows if the number of rows needed is low in proportion to the total number of rows in the table and if the table is large. Consider this statement:

```
select count(*) from employees where last_name between 'A%' and 'Z%';
```

This WHERE clause is sufficiently broad that it will include every row in the table. It would be much slower to such the index to find the rowids and then use the rowids to find the rows than to scan the whole table. After all, it is the whole table that is needed. Another example would be if the table were small enough that one disk read could scan it in its entirety: there would be no point in reading an index first.

It is often said that if the query is going to retrieve more than 2 to 4 percent of the rows, then a full table scan will be quicker. A major exception to this is if the value specified in the WHERE clause is NULL. NULLs do not go into B*Tree indexes, so a query such as this:

```
select * from employees where last_name is null;
```

will always result in a full table scan. There is little value in creating a B*Tree index on a column with few unique values, as it will not be selective: the proportion of the table that will be retrieved for each distinct key value will be too high. In general, B*Tree indexes should be used if:

- The cardinality (the number of distinct values) in the column is high, and
- The number of rows in the table is high, and
- The column is used in WHERE clauses or JOIN conditions

The B*Tree structure is very efficient. If the depth is greater then three or four, than either the index keys are very long or the table has billions of rows. If neither if these is the case, then the index is in need of a rebuild.

Bitmap Indexes

In many business applications, the nature of the data and the queries is such that B*Tree indexes are not of much use. Consider the table of sales for a chain of supermarkets, storing one year of historical data, which can be analyzed in several dimensions.

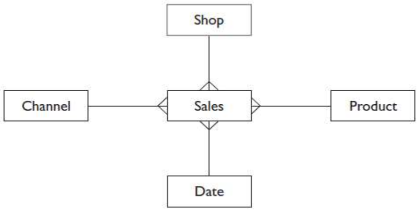


Figure shows a simple entity-relationship diagram, with just four of the dimensions. The cardinality of each dimension could be quite low. Make these assumptions:

SHOP	There are four shops
PRODUCT	There are 200 products
DATE	There are 365 days
CHANNEL	There are two channels (walk-in and delivery)

Assuming an even distribution of data, only two of the dimensions (PRODUCT and DATE) have a selectivity better than the commonly used criterion of 2 percent to 4 percent that makes an index worthwhile. But if queries use range predicates (such as counting sales in a month, or of a class of ten or more products) then not even these will qualify. This is a simple fact: B*Tree indexes are often useless in a data warehouse environment. A typical query might want to compare sales between two shops to walk-in customers of a certain class of product in a month. There may well be B*Tree indexes on the relevant columns, but Oracle will ignore them as being insufficiently selective. This is what bitmap indexes are designed for. A bitmap index stores the rowids associated with each key value as a bitmap. The
bitmaps for the CHANNEL index might look like this:

```
WALKIN 11010111000101011101011101…..
DELIVERY 00101000111010100010100010…..
```

This indicates that the first two rows were sales to walk-in customers, the third sale was a delivery, the fourth sale was a walk-in, and so on.
The bitmaps for the SHOP index might be:

```
LONDON 11001001001001101001010000…..
OXFORD 00100010011000010001001000…..
READING 00010001000100000100100010…..
GLASGOW 000001001000100000010000101…..
```

This indicates that the first two sales were in the London shop, the third was in Oxford, the fourth in Reading, and so on. Now if this query is received:

```
select count(*) from sales where channel='WALKIN' and shop='OXFORD';
```

Oracle can retrieve the two relevant bitmaps and add them together with a Boolean AND operation:

```
WALKIN 11010111000101011101011101…..
OXFORD 00100010011000010001001100…..
WALKIN & OXFORD 0000001000000000100000000000…..
```

The result of the AND operation shows that only the seventh and sixteenth rows qualify for selection. This combining of bitmaps is very fast and can be used to implement complex Boolean algebra operations with many conditions on many columns using any combination of AND, OR, and NOT operators. A particular advantage that bitmap indexes have over B*Tree indexes is that they include NULLs. As far as the bitmap index is concerned, NULL is just another distinct value, which will have its own bitmap.
In general, bitmap indexes should be used if:

- The cardinality (the number of distinct values) in the column is low (such as male/female), and
- The number of rows in the table is high, and
- The column is used in Boolean algebra (AND/OR/NOT) operations

If you knew in advance what the queries would be then you could build B*Tree indexes that would work, such as a composite index on SHOP and CHANNEL. But usually you don't know, which is where the dynamic merging of bitmaps gives great flexibility.

Creating and Using Indexes

Indexes are created implicitly when primary key and unique constraints are defined, if an index on the relevant column(s) does not already exist.
The basic syntax for creating an index explicitly is as follows:
CREATE [UNIQUE | BITMAP] INDEX [*schema.*]*indexname*
ON [*schema.*]*tablename (column* [, *column*…]) ;
The default type of index is a nonunique B*Tree index. It is not possible to create a unique bitmap index (and you wouldn't want to if you could: think about the cardinality issue). Indexes are schema objects, and it is possible to create an index in one schema on a table in another, but most people would find this somewhat confusing.
A *composite* index is an index on several columns. Composite indexes can be on columns of different data types, and the columns do not have to be adjacent in the table.

Consider this example of creating tables and indexes and then defining constraints:

```
create table dept(deptno number,dname varchar2(10));
create table emp(empno number, surname varchar2(10), forename varchar2(10), dob
date, deptno number);
create unique index dept_i1 on dept(deptno);
create unique index emp_i1 on emp(empno);
create index emp_i2 on emp(surname,forename);
create bitmap index emp_i3 on emp(deptno);
alter table dept add constraint dept_pk primary key (deptno);
alter table emp add constraint emp_pk primary key (empno);
alter table emp add constraint emp_fk foreign key (deptno) references
dept(deptno);
```

The first two indexes created are flagged as UNIQUE, meaning that it will not be possible to insert duplicate values. This is not defined as a constraint at this point but is true nonetheless. The third index is not defined as UNIQUE and will therefore accept duplicate values; this is a composite index on two columns. The fourth index is defined as a bitmap index, because the cardinality of the column is likely to be low in proportion to the number of rows in the table.
When the two primary key constraints are defined, Oracle will detect the precreated indexes and use them to enforce the constraints. Note that the index on DEPT.DEPTNO has no purpose for performance because the table will in all likelihood be so small that the index will never be used to retrieve rows (a scan will be quicker), but it is still essential to have an index to enforce the primary key constraint.
Once created, use of indexes is completely transparent and automatic. Before executing a SQL statement, the Oracle server will evaluate all the possible ways of executing it. Some of these ways may involve using whatever indexes are available, others may not. Oracle will make use of the information it gathers on the tables and the environment to make an intelligent decision about which (if any) indexes to use.
A unique and primary key constraint can be enforced by indexes that are either unique or nonunique: in the latter case, it will be a nonunique index that happens to have only unique values.

The Oracle server should make the best decision about index use, but if it gets it wrong it is possible for a programmer to embed instructions, known as optimizer hints, in code that will force the use (or not) of certain indexes.

Modifying and Dropping Indexes

There is a command, ALTER INDEX…but it cannot be used to change any of the characteristics described in this chapter: the type (B*Tree or bitmap) of the index, the columns, or whether it is unique or nonunique. The ALTER INDEX command lies in the database administration domain and would typically be used to adjust the physical properties of the index, not the logical properties that are of interest to developers. If it is necessary to change any of these properties, the index must be dropped and recreated.

Continuing the example in the previous section, to change the index EMP_I2 to include the employees' birthdays:

```
drop index emp_i2;
create index emp_i2 on emp(surname,forename,dob);
```

This composite index now includes columns with different data types. The columns happen to be listed in the same order that they are defined in the table, but this is by no means necessary.

When a table is dropped, all the indexes and constraints defined for the table are dropped as well. If an index was created implicitly by creating a constraint, then dropping the constraint will also drop the index. If the index had been created explicitly and the constraint created later, then if the constraint is dropped the index will survive.

Creating Indexes

In this exercise, create indexes on a copy of the EMPLOYEES table in the HR schema. Either SQL*Plus or SQL Developer can be used.

1. Connect to your database as user HR.
2. Create a table that is a copy of EMPLOYEES:

```
create table emps as select * from employees;
```

This table will have neither indexes nor primary, unique, or foreign key constraints, because these are not copied by a CREATE TABLE AS command. The NOT NULL constraints will have been copied. Confirm this by describing the table:

```
describe emps;
```

3. Create an index to be used for the primary key constraint:

```
create unique index emps_empid_i on emps(employee_id);
```

4. Demonstrate that a unique index cannot accept duplicates, even before a constraint is defined:

```
insert into emps(employee_id,last_name,email,hire_date,job_id)
values(198,'Watson','jw@bplc.co.za',sysdate,'IT_PROG');
```

This will return an error because the index cannot insert a second employee_id 198. Index uniqueness is an attribute of the index that can exist without a constraint but should not be relied upon to enforce data integrity.

5. Create additional indexes on columns that are likely to be used in WHERE clauses, using B*Tree for columns of high cardinality and bitmap for columns of low cardinality:

```
create index emps_name_i on emps(last_name,first_name);
create index emps_tel_i on emps(phone_number);
create bitmap index emps_mgr_i on emps(manage_id);
create bitmap index emps_dept_i on emps(department_id);
```

6. Define some constraints:

```
alter table emps add constraint emps_pk primary key (employee_id);
alter table emps add constraint emps_email_uk unique(email);
alter table emps add constraint emps_tel_uk unique(phone_number);
```

7. Display the index names and their type:

```
select index_name,index_type,uniqueness from user_indexes
where table_name='EMPS';
```

The view USER_INDEXES shows details of all indexes in your current schema.

Note that in addition to the five indexes explicitly created in steps 3 and 5, there is also an index created implicitly with the name of the constraint defined on EMAIL. Note also that the unique constraint on PHONE_NUMBER is being enforced with a nonunique index; this is perfectly possible, because although the constraint mechanism uses indexes, it is independent of the structure of the index.

8. Tidy up by dropping the EMPS table, and confirm that all the indexes have also gone:

```
drop table emps;
select index_name from user_indexes where table_name='EMPS';
```

TWO-MINUTE DRILL

Create Simple and Complex Views

❑ A simple view has one detail (or base) table and uses neither functions nor aggregation.

❑ A complex view can be based on any SELECT statement, no matter how complicated.

❑ Views are schema objects. To use a view in another schema, the view name must be qualified with the schema name.

Retrieve Data from Views

❑ A view can be queried exactly as though it were a table.

❑ Views can be joined to other views or to tables, they can be aggregated, and in some cases they can accept DML statements.

❑ Views exist only as data dictionary constructs. Whenever you query a view, the underlying SELECT statement must be run.

Create Private and Public Synonyms

❑ A synonym is an alternative name for a view or a table.

❑ Private synonyms are schema objects; public synonyms exist outside user schemas and can be used without specifying a schema name as a qualifier.

❑ Synonyms share the same namespace as views and tables and can therefore be used interchangeably with them.

Create, Maintain, and Use Sequences

❑ A sequence generates unique values—unless either MAXVALUE or MINVALUE and CYCLE have been specified.

❑ Incrementing a sequence need not be committed and cannot be rolled back.

❑ Any session can increment the sequence by reading its next value. It is possible to obtain the last value issued to your session but not the last value issued.

Create and Maintain Indexes

❑ Indexes are required for enforcing unique and primary key constraints.

❑ NULLs are not included in B*Tree indexes but are included in bitmap indexes.

❑ B*Tree indexes can be unique or nonunique, which determines whether they can accept duplicate key values.

❑ B*Tree indexes are suitable for high cardinality columns, bitmap indexes for low cardinality columns.

❑ Compound indexes have a key consisting of several columns, which can be of different data types.

SELF TEST

Choose all the correct answers for each question.

Create Simple and Complex Views

1. Which of these is a defining characteristic of a complex view, rather than a simple view? (Choose one or more correct answers.)

A. Restricting the projection by selecting only some of the table's columns

B. Naming the view's columns with column aliases

C. Restricting the selection of rows with a WHERE clause

D. Performing an aggregation

E. Joining two tables

2. Consider these three statements:

```
create view v1 as select department_id,department_name,last_name from
departments join employees using (department_id);
select department_name,last_name from v1 where department_id=20;
select d.department_name,e.last_name from departments d, employees e
where d.department_id=e.department_id and
d.department_id=20;
```

The first query will be quicker than the second because (choose the best answer):

A. The view has already done the work of joining the tables.

B. The view uses ISO standard join syntax, which is faster than the Oracle join syntax used in the second query.

C. The view is precompiled, so the first query requires less dynamic compilation than the second query.

D. There is no reason for the first query to be quicker.

3. Study this view creation statement:

```
create view dept30 as
select department_id,employee_id,last_name from employees
where department_id=30 with check option;
```

What might make the following statement fail? (Choose the best answer.)

```
update dept30 set department_id=10 where employee_id=114;
```

A. Unless specified otherwise, views will be created as WITH READ ONLY.

B. The view is too complex to allow DML operations.

C. The WITH CHECK OPTION will reject any statement that changes the DEPARTMENT_ID.

D. The statement will succeed.

Retrieve Data from Views

4. There is a simple view SCOTT.DEPT_VIEW on the table SCOTT.DEPT. This insert fails with an error:

```
SQL> insert into dept_view values('SUPPORT','OXFORD');
insert into dept_view values('SUPPORT','OXFORD')
*
ERROR at line 1:
ORA-01400: cannot insert NULL into ("SCOTT"."DEPT"."DEPTNO")
```

What might be the problem? (Choose the best answer.)

A. The INSERT violates a constraint on the detail table.

B. The INSERT violates a constraint on the view.

C. The view was created as WITH READ ONLY.

D. The view was created as WITH CHECK OPTION.

Create Private and Public Synonyms

5. What are distinguishing characteristics of a public synonym rather than a private synonym? (Choose two correct answers.)

A. Public synonyms are always visible to all users.

B. Public synonyms can be accessed by name without a schema name qualifier.

C. Public synonyms can be selected from without needing any permissions.

D. Public synonyms can have the same names as tables or views.

6. Consider these three statements:

```
create synonym s1 for employees;
create public synonym s1 for departments;
select * from s1;
```

Which of the following statements is correct? (Choose the best answer.)

A. The second statement will fail because an object S1 already exists.

B. The third statement will show the contents of EMPLOYEES.

C. The third statement will show the contents of DEPARTMENTS.

D. The third statement will show the contents of the table S1, if such a table exists in the current schema.

7. A view and a synonym are created as follows:

```
create view dept_v as select * from dept;
create synonym dept_s for dept_v;
```

Subsequently the table DEPT is dropped. What will happen if you query the synonym DEPT_S ? (Choose the best answer.)

A. There will not be an error because the synonym addresses the view, which still exists, but there will be no rows returned.

B. There will not be an error if you first recompile the view with the command ALTER VIEW DEPT_V COMPILE FORCE;

C. There will be an error because the synonym will be invalid.

D. There will be an error because the view will be invalid.

E. There will be an error because the view will have been dropped implicitly when the table was dropped.

Create, Maintain, and Use Sequences

8. A sequence is created as follows:

```
create sequence seq1 maxvalue 50;
```

If the current value is already 50, when you attempt to select SEQ1.NEXTVAL what will happen? (Choose the best answer.)

A. The sequence will cycle and issue 0.

B. The sequence will cycle and issue 1.

C. The sequence will reissue 50.

D. There will be an error.

9. You create a sequence as follows:

```
create sequence seq1 start with 1;
```

After selecting from it a few times, you want to reinitialize it to reissue the numbers already

generated. How can you do this? (Choose the best answer.)
A. You must drop and re-create the sequence.
B. You can’t. Under no circumstances can numbers from a sequence be reissued once they have been used.
C. Use the command ALTER SEQUENCE SEQ1 START WITH 1; to reset the next value to 1.
D. Use the command ALTER SEQUENCE SEQ1 CYCLE; to reset the sequence to its starting value.
10. Study the following exhibit:
Assuming that the sequence SEQ1 was created with the option ORDER and INCREMENT BY set to 1, what value will be returned by the final SELECT statement? (Choose the best answer.)
A. 2
B. 3
C. 4
D. It will depend on whether any other sessions are selecting from the sequence while the statements in the exhibit are being run.

Create and Maintain Indexes

11. A UNIQUE constraint on a column requires an index. Which of the following scenarios is correct? (Choose one or more correct answers.)
A. If a UNIQUE index already exists on the column, it will be used.
B. If a NONUNIQUE index already exists it will be used.
C. If a NONUNIQUE index already exists on the column, a UNIQUE index will be created implicitly.
D. If any index exists on the column, there will be an error as Oracle attempts to create another index implicitly.
12. This statement will fail:

```
create unique bitmap index on employees(department_id,hire_date);
```

Why? (Choose the best answer.)
A. Bitmap indexes cannot be unique.
B. The two columns are of different data types.
C. A bitmap index can be on only one column.
D. There is already a B*Tree index on DEPARTMENT_ID.
13. You have created an index with this statement:

```
create index ename_i on employees(last_name,first_name);
```

How can you adjust the index to include the employees’ birthdays, which is a date type column called DOB? (Choose the best answer.)
A. Use ALTER INDEX ENAME_I ADD COLUMN DOB;.
B. You can’t do this because of the data type mismatch.
C. You must drop the index and re-create it.
D. This can only be done if the column DOB is NULL in all existing rows.

LAB QUESTION

The columns for the fact table SALES are as follows:

- **SALE_ID** System-generated primary key
- **CHANNEL_ID** Foreign key to CHANNELS
- **PRODUCT_ID** Foreign key to PRODUCTS
- **SHOP_ID** Foreign key to SHOPS
- **DAY_ID** Foreign key to DAYS
- **QUANTITY** The quantity of the product sold

It is expected that there will be several million SALES rows per year. The dimension tables are as follows:
PRODUCTS A list of all products, including price. Cardinality of a few hundred **CHANNEL** Possible sales methods, such as walk-in, Internet, and telephone **SHOPS** Details of all the shops—no more that a couple of dozen **DAYS** Dates for which sales are being stored: 365, identified by day number.
Write code to create the tables; create indexes; create constraints. Create sequences to be used for primary keys where necessary. Create some views that will present the data in an easy-to-understand fashion.

SELF TEST ANSWERS

Create Simple and Complex Views

1. ✹✓ D, E. Aggregations and joins make a view complex and make DML impossible.

✹➢ **A, B, C.** Selection and projection or renaming columns does not make the view complex.

2. ✹✓ D. Sad but true. Views do not help performance.

✹➢ **A** is wrong because a view is only a SELECT statement; it doesn’t prerun the query. **B** is wrong because the Oracle optimizer will sort out any differences in syntax. **C** is wrong because, although views are precompiled, this doesn’t affect the speed of compiling a user’s statement.

3. ✹✓ C. The WITH CHECK OPTION will prevent DML that would cause a row to disappear from the view.

✹➢ **A, B, D. A** is wrong because views are by default created read/write. **B** is wrong because the view is a simple view. **D** is wrong because the statement cannot succeed because the check option will reject it.

Retrieve Data from Views

4. ✹✓ A. There is a NOT NULL or PRIMARY KEY constraint on DEPT.DEPTNO.

✹➢ **B, C, D. B** is wrong because constraints are enforced on detail tables, not on views. **C** and **D** are wrong because the error message would be different.

Create Private and Public Synonyms

5. ✹✓ B, D. Public synonyms are not schema objects and so can only be addressed directly. They can have the same names as schema objects.

✹➢ **A, C.** These are wrong because users must be granted privileges on a public synonym before they can see it or select from it.

6. ✹✓ B. The order of priority is to search the schema namespace before the public namespace, so it will be the private synonym (to EMPLOYEES) that will be found.

✹➢ **A, C, D. A** is wrong because a synonym can exist in both the public namespace and the schema namespace. **C** is wrong because the order of priority will find the private synonym first. **D** is wrong because it would not be possible to have a table and a private synonym in the same schema with the same name.

7. ✹✓ D. The synonym will be fine, but the view will be invalid. Oracle will attempt to recompile the view, but this will fail.

✹➢ **A, B, C, E. A** is wrong because the view will be invalid. **B** is wrong because the FORCE keyword can only be applied when creating a view (and it would still be invalid, even so). **C** is wrong because the synonym will be fine. **E** is wrong because views are not dropped implicitly (unlike indexes and constraints).

Create, Maintain, and Use Sequences

8. ✹✓ **D.** The default is NOCYCLE, and the sequence cannot advance further.

✹✗ **A, B, C. A** and **B** are wrong because CYCLE is disabled by default. If it were enabled, the next number issued would be 1 (not zero) because 1 is the default for START WITH. **C** is wrong because under no circumstances will a sequence issue repeating values.

9. ✹✓ **A.** It is not possible to change the next value of a sequence, so you must re-create it.

✹✗ **B, C, D. B** is wrong because, while a NOCYCLE sequence can never reissue numbers, there is no reason why a new sequence (with the same name) cannot do so. **C** is wrong because START WITH can only be specified at creation time. **D** is wrong because this will not force an instant cycle; it will only affect what happens when the sequence reaches its MAXVALUE or MINVALUE.

10. ✹✓ **D.** If the sequence is being used by other sessions, there is no knowing how many increments may have taken place between the first and the second INSERT statements.

✹✗ **A, B, C.** The answer would be 4, **C**, except that there could have been increments forced by other sessions. **A** and **B** are wrong because the ROLLBACK will not reverse the sequence increments.

Create and Maintain Indexes

11. ✹✓ **A, B.** Either a UNIQUE or a NONUNIQUE index can be used to enforce a UNIQUE constraint.

✹✗ **C, D. C** is wrong because there is no need to create another index (in fact, you can't index the same column twice even if you want to). **D** is wrong because if an index exists, Oracle won't attempt to create another.

12. ✹✓ **A.** The keywords BITMAP and UNIQUE are mutually exclusive. And you wouldn't want to do this, anyway.

✹✗ **B, C, D. B** and **C** are wrong because a bitmap index can be composite, with columns of different data types. **D** is wrong because the bitmap index is not on DEPARTMENT_ID alone, which would not be possible.

13. ✹✓ **C.** It is not possible to change an index's columns after creation.

✹✗ **A, B, D. B** is wrong because the data type is not the problem. **A** and **D** are wrong because an index's columns are fixed at creation time.

LAB ANSWER

This is a possible solution:

```
/*create the tables*/
create table sales (sale_id number, channel_id number, product_id number, shop_id number,
day_id number, quantity number);
create table products (product_id number, pname varchar2(20),price number);
create table channels (channel_id number, cname varchar2(20));
create table shops (shop_id number,address varchar2(20));
create table days (day_id number, day date);
/*pre-create indexes to be used for constraints*/
create unique index prod_pk on products(product_id);
create unique index chan_pk on channels(channel_id);
create unique index shop_pk on shops(shop_id);
create unique index day_id on days(day_id);
create unique index sales_pk on sales(sale_id);
/*create bitmap indexes on the dimension columns of the fact table*/
create bitmap index sales_chan on sales(channel_id);
create bitmap index sales_prod on sales(product_id);
create bitmap index sales_shop on sales(shop_id);
create bitmap index sales_date on sales(day_id);
/*add the primary key constraints/*
alter table products add constraint prod_pk primary key (product_id);
alter table channels add constraint chan_pk primary key (channel_id);
alter table shops add constraint shop_pk primary key (shop_id);
alter table days add constraint day_pk primary key (day_id);
alter table sales add constraint sales_pk primary key(sale_id);
/*add the foreign key constraints*/
alter table sales add constraint sales_prod_fk foreign key (product_id) references products;
alter table sales add constraint sales_chan_fk foreign key (channel_id) references channels;
alter table sales add constraint sales_shop_fk foreign key (shop_id) references shops;
alter table sales add constraint sales_day_fk foreign key (day_id) references days;
/*create the sequences for primary keys:
cache many values for the fact table, but don't pre-issue values for the largely static
dimension tables. This will save some memory*/
create sequence sales_seq cache 1000;
create sequence product_seq nocache;
create sequence channel_seq nocache;
create sequence shop_seq nocache;
create sequence day_seq nocache;
/*create a view to analyze sales in several dimensions*/
create view sales_analysis as
select cname,pname,address,day,sum(quantity) total
from sales,channels,products,shops,days
where sales.channel_id=channels.channel_id
and sales.product_id=products.product_id
and sales.shop_id=shops.shop_id
and sales.day_id=days.day_id
group by grouping sets(
(cname,pname,address,day),
(address,pname),
(pname,day));
```