Contents

Writing Basic SQL SELECT Statements .............................................................................................................. 6
List the Capabilities of SQL SELECT Statements ............................................................................................... 7
Execute a Basic SELECT Statement .................................................................................................................... 9
Differentiate Between SQL Statements and /SQL*Plus Commands ................................................................. 14
Review Checklist: Writing Basic SQL SELECT Statements .................................................................................. 15
Restricting and Sorting Data ............................................................................................................................ 16
Limit the Rows Retrieved by a Query .................................................................................................................. 17
Sort the Rows Retrieved by a Query ................................................................................................................... 24
Review Checklist: Restricting and Sorting Data .................................................................................................. 26
Single-Row Functions ....................................................................................................................................... 27
Describe the Various Types of Functions Available in SQL ............................................................................. 28
Use Character, Number, and Date Functions in SQL ......................................................................................... 32
Use Conversion Functions ................................................................................................................................... 37
Review Checklist: Single-Row Functions ............................................................................................................. 40
Displaying Data from Multiple Tables ............................................................................................................... 41
Write SELECT Statements to Access Data from More than One Table Using Equality and Nonequality Joins ................................................................................................................................. 42
View Data that Generally Does Not Meet a Join Condition by Using Outer Joins ........................................... 47
Join a Table to Itself in Self-Join ......................................................................................................................... 50
Review Checklist: Displaying data from Multiple Tables .................................................................................... 51
Aggregating Data Using Group Functions ......................................................................................................... 52
Identifying the Available Group Functions ......................................................................................................... 53
Use Group Functions .......................................................................................................................................... 55
Group Data Using GROUP BY Clause ................................................................................................................ 58
Include or Exclude Grouped Rows by Using the HAVING Clause .................................................................... 60
Review Checklist: Aggregating Data Using Group Functions ........................................................................... 62
Subqueries ............................................................................................................................................................ 63
Describe the Types of Problems that Subqueries Can Solve ............................................................................ 64
Define Subqueries and Write Single-Row and Multiple-Row Subqueries ......................................................... 67
List the Types of Subqueries ............................................................................................................................. 73
Review Checklist: Subqueries ............................................................................................................................ 76
Producing Readable Output with iSQL*Plus ...................................................................................................... 77
Produce Queries that Require a Substitution Variable ...................................................................................... 78
Produce Readable Output .................................................................................................................................. 82
Create and Execute Script Files ......................................................................................................................... 86
Review Checklist: Producing Readable Output with iSQL*Plus ...................................................................... 89
Manipulating Data ............................................................................................................................................. 90
Describe Each DML Statement .......................................................................................................................... 91
Insert Rows into a Table ..................................................................................................................................... 93
Update Rows in a Table ...................................................................................................................................... 96
Delete Rows from a Table ............................................................................................................................... 98
Merge Rows in a Table .................................................................................................................................... 100
Control Transactions ................................................................................................................................. 101
Review Checklist: Manipulating Data............................................................................................................. 104
**Creating and Managing Tables** .............................................................................................................. **105**
Describe the Main Database Objects ............................................................................................................. 106
Create Tables .................................................................................................................................................. 107
Describe the Data Types that Can be Used when Specifying Column Definitions....................................... 111
Alter Table Definitions ................................................................................................................................... 115
Drop, Rename, and Truncate Tables ................................................................................................................ 118
Review Checklist: Creating and Managing Tables .......................................................................................... 119
**Including Constraints** .............................................................................................................................. **120**
Describe Constraints ...................................................................................................................................... 121
Create and Maintain Constraints ..................................................................................................................... 123
Review Checklist: Including Constraints .......................................................................................................... 129
**Creating Views** ........................................................................................................................................ **130**
Describe a View ............................................................................................................................................ 131
Create, Alter the Definition, and Drop a View ................................................................................................. 132
Retrieve Data through a View ............................................................................................................................ 136
Insert, Update, and Delete Data through a View ............................................................................................ 139
Create and Use an Inline View .......................................................................................................................... 143
Perform Top 'N' Analysis ................................................................................................................................. 144
Review Checklist: Creating Views .................................................................................................................... 146
**Creating Other Database Objects** ............................................................................................................ **147**
Create, Maintain, and Use Sequences ............................................................................................................... 148
Create and Maintain Indexes ............................................................................................................................. 151
Create Private and Public Synonyms ................................................................................................................ 153
Review Checklist: Creating Other Database Objects .......................................................................................... 154
**Controlling User Access** ........................................................................................................................... **155**
Create Users .................................................................................................................................................. 156
Create Roles to Ease the Setup and Maintenance of the Security Model ..................................................... 158
Use the GRANT and REVOKE Statements to Grant and Revoke Object Privileges ....................................... 159
Review Checklist: Controlling User Access ..................................................................................................... 162
**Test Taking Strategies** ............................................................................................................................. **163**
Thank You and Good Luck on your Exams! ..................................................................................................... 166
Pass the Exam with the Self Test Study Guide

As a certification candidate, you’re on the lookout for as many test preparation resources as possible, but your time is at a premium. So, we put together the Study Guide with your study goals and busy schedule in mind.

Create a study plan – and stick to it. Don’t try to cram. Set aside specific study times so you can thoroughly prepare for your exam. In our experience, following a disciplined prep schedule leads to success on Exam Day. To thoroughly gain the benefits of our Study Guide, we recommend that you start preparing about six weeks prior to taking your exam.

Use all the prep tools in your Study Guide. Your Guide contains a full suite of products to help you thoroughly prepare for your exam. Each one is designed with a specific study purpose in mind. Here’s how we recommend you use the products together:

1. Perform a baseline checkup. Set your personal baseline by taking the Self Test Practice Test in Certification Mode. It is a timed test that simulates the real exam. Objective-based scoring shows you the areas you are relatively strong in, and those you need to devote additional time to.

2. Concentrate your studies by objective. Since your study time has to be in chunks, use the exam’s structure by objective to help you organize your study sessions. Use each product to study at the objective level with particular emphasis on the objectives where you did not score 100% in your baseline checkup.
   - **Study Guide.** Read the Study Guide by objective to familiarize yourself with the exam content. The Study Guide is objective-driven and contains a Scope and Focused Explanation for each objective. At the end of each major objective is a Review Checklist that lists the key points covered in this area of the exam.
   - **Self Test Flash Cards.** Drill through the Flash Cards by objective to be sure you know the fundamental concepts. Make Personal Study Notes with these cards to supplement your learning.
   - **Self Test Practice Test.** Use the Practice Test in Learning Mode by objective. Answer the questions, read the tutorials, and use the Personal Study Notes to supplement your learning and/or highlight items that you’ll want to review before the exam.
   - **References.** Use your favorite references (books, web references, etc.) to get additional materials on more complex subject matter.

3. Track your progress. You’ve completed your objective-driven study plan. Now you’re ready to see how you’ve progressed. Take the Self Test Practice Test in Certification Mode again. Did you score 100%? If not, go back to your objective study plan and focus on your weaknesses. Keep checking yourself, highlighting objectives that you’ll need to study, until you consistently score 100%.

Do your final preparation. Print the Review Checklists from the Study Guide and the Personal Study Notes from the Self Test Practice Test and Flash Cards. Use these as your condensed final review before taking the real exam. And, before taking the real exam, read the Test-Taking Strategies at the end of this guide to get specific techniques on approaching the different question types you may encounter.

Finally, some last words of advice. During your studies and practice test drills, concentrate on the process - not totally on performance. What matters most is that you are using a disciplined approach that covers the materials on the exam and you know what to expect on exam day. Stay the course of your study plan and you will be ready to PASS THE EXAM!
Writing Basic SQL SELECT Statements
List the Capabilities of SQL SELECT Statements

Scope

Understand and define the purpose of a SELECT statement. Know the definitions of selection, projection, and join. Know and be able to identify the proper syntax for a basic SELECT statement.

Focused Explanation

A SQL SELECT statement retrieves data from the database, allowing you to display data stored in a database table. Using a SELECT statement, you can do the following:

Projection – Using the projection capability, you can specify columns in a table that you want returned by the query. Projection is often referred to as vertical partitioning.

```
SELECT stu_name, tuition FROM student;
```

<table>
<thead>
<tr>
<th>STUDENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>STU_ID</td>
</tr>
<tr>
<td>---------</td>
</tr>
</tbody>
</table>

Selection – Using the selection capability, you can specify the rows in a table that you want returned by the query. You can specify criteria in your SELECT statement to restrict the rows that are returned. Selection is often referred to as horizontal partitioning.

```
SELECT * FROM student WHERE tuition > 50000;
```

<table>
<thead>
<tr>
<th>STUDENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>STU_ID</td>
</tr>
<tr>
<td>---------</td>
</tr>
</tbody>
</table>

Join – Using the join capability, you can bring together data that is stored in different tables. In the SELECT statement, you create a link between the tables to retrieve data from each table.

The basic SELECT statement syntax:

```
SELECT * | { [DISTINCT] column | expression [alias] , . . . } FROM table_name;
```
In the syntax:

- **SELECT** identifies the columns to be displayed
- * selects all columns to be displayed
- **DISTINCT** suppresses duplicate values
- **column | expression** specifies the named column or the expression
- **alias** gives the selected columns different headings
- **FROM table_name** specifies the table containing the selected columns
Execute a Basic SELECT Statement

Scope

Perform basic calculations in a SELECT statement using the arithmetic operators. Specify column aliases in a SELECT clause and identify the proper syntax for column aliases. Use the concatenate operator to combine columns and literals in SELECT statements. Specify columns for display using the SELECT clause. Suppress duplicate output using the DISTINCT keyword.

Focused Explanation

To modify the way in which number and date data is displayed, you can use arithmetic expressions. The arithmetic operators:

```
SELECT stu_name, tuition, tuition - 450
FROM student;
```

```
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Add</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
</tr>
</tbody>
</table>
```

```
STUDENT

<table>
<thead>
<tr>
<th>STU_NAME</th>
<th>TUITION</th>
<th>TUITION-450</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>4500</td>
<td>4050</td>
</tr>
</tbody>
</table>
```

Null values in calculations result in a null value.

```
SELECT stu_name, tuition, tuition / 2
FROM student;
```

```
STUDENT

<table>
<thead>
<tr>
<th>STU_NAME</th>
<th>TUITION</th>
<th>TUITION/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

This is the order of operator precedence:

\[
\ast \quad / \quad + \quad - \quad _
\]

The multiplication and division operators are evaluated before the addition and subtraction operators. If operators have the same priority in a calculation, they are evaluated from left to right.
Parentheses are used in calculations to prioritize evaluation of operators and for clarity.

```
SELECT stu_name, tuition, 4 * (tuition - 100)
FROM student;
```

<table>
<thead>
<tr>
<th>STUDENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>STU_NAME</td>
</tr>
<tr>
<td>Jones</td>
</tr>
</tbody>
</table>

In this example, the parentheses override the rule of precedence, and the addition operator is evaluated first. The calculation is performed as follows:

\[
5500 - 100 = 5400 \\
5400 \times 4 = 21600
\]

This is the statement and output without parentheses:

```
SELECT stu_name, tuition, 4 * tuition - 100
FROM student;
```

<table>
<thead>
<tr>
<th>STUDENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>STU_NAME</td>
</tr>
<tr>
<td>Jones</td>
</tr>
</tbody>
</table>

The calculation is performed as follows:

\[
4 \times 5500 = 22000 \\
22000 - 100 = 21900
\]

In this code example, the parentheses have no effect on the output returned:

```
SELECT stu_name, tuition, (4 * tuition) - 100
FROM student;
```

A column alias renames a column heading and immediately follows the column name in the SELECT clause. You can use the optional AS keyword when specifying a column alias.
These two SELECT statements result in the same output:

```
SELECT stu_name AS name  SELECT stu_name name
FROM student;   FROM student;
```

<table>
<thead>
<tr>
<th>STUDENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
</tr>
<tr>
<td>Jones</td>
</tr>
</tbody>
</table>

By default, a column alias heading is displayed in uppercase. To display a case sensitive column heading, enclose the column alias in double quotation marks (" "). Use this same method if the column alias contains special characters or spaces.

```
SELECT stu_name AS "Student Name", stu_name "Name"
FROM student;
```

<table>
<thead>
<tr>
<th>STUDENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student Name</td>
</tr>
<tr>
<td>Jones</td>
</tr>
</tbody>
</table>

To select all columns of data in a table, follow the SELECT keyword with an asterisk ( * ) or list all the columns after the SELECT keyword, separated by commas.

```
SELECT *
FROM student;
```

```
SELECT stu_id, stu_name, adm_date, tuition, major_id
FROM student;
```

You can display specific columns from a table by specifying the column names and separating each column name with a comma.

```
SELECT stu_name, major_id
FROM student;
```
The default output of queries includes duplicate rows:

```sql
SELECT major_id
FROM student;
```

<table>
<thead>
<tr>
<th>MAJOR_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>20</td>
</tr>
</tbody>
</table>

You can include the DISTINCT keyword in the SELECT clause immediately after the SELECT keyword to eliminate duplicate rows in the output.

```sql
SELECT DISTINCT major_id
FROM student;
```

<table>
<thead>
<tr>
<th>MAJOR_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>30</td>
</tr>
</tbody>
</table>

If you specify multiple columns after the DISTINCT keyword, all selected columns are affected and the output is a distinct combination of all the specified columns.

```sql
SELECT DISTINCT major_id, stu_name
FROM student;
```

<table>
<thead>
<tr>
<th>MAJOR_ID</th>
<th>STU_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>JONES</td>
</tr>
<tr>
<td>10</td>
<td>BROWN</td>
</tr>
<tr>
<td>15</td>
<td>SMITH</td>
</tr>
<tr>
<td>20</td>
<td>BROWN</td>
</tr>
<tr>
<td>20</td>
<td>SMITH</td>
</tr>
</tbody>
</table>
Using the concatenation operator (||), you can link columns to other columns to create a character expression. The concatenation operator combines columns on either side of the operator to make a single output column. You can also use the concatenate to link arithmetic expressions or constant values to create a character expression.

```
SELECT stu_name||major_id AS “Student ID”
FROM student;
```

<table>
<thead>
<tr>
<th>Student ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones10</td>
</tr>
<tr>
<td>Brown20</td>
</tr>
<tr>
<td>Smith15</td>
</tr>
</tbody>
</table>

You must enclose date and literal character strings within single quotation marks (‘‘).

```
SELECT stu_name || ‘ owes ’ || tuition “Tuition Due”
FROM student;
```

<table>
<thead>
<tr>
<th>TUITION DUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones owes 5500</td>
</tr>
<tr>
<td>Brown owes 4500</td>
</tr>
<tr>
<td>Smith owes 5200</td>
</tr>
</tbody>
</table>
Differentiate Between SQL Statements and iSQL*Plus Commands

**Scope**
Understand the differences between SQL and iSQL*Plus and how to use iSQL*Plus, including the iSQL*Plus DESCRIBE command.

**Focused Explanation**

SQL statements versus iSQL*Plus commands:

SQL is a command language. Oracle uses SQL to communicate with the Oracle server from any tool or application.

iSQL*Plus is an Oracle tool. iSQL*Plus submits SQL statements to the Oracle server for execution. iSQL*Plus contains its own command language. iSQL*Plus runs on a browser and is centrally loaded. It does not need to be implemented on each machine.

<table>
<thead>
<tr>
<th>SQL</th>
<th>iSQL*Plus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is a language</td>
<td>Is an environment</td>
</tr>
<tr>
<td>Is an ANSI standard</td>
<td>Is an Oracle proprietary tool</td>
</tr>
<tr>
<td>SQL statements manipulate database data and table definitions</td>
<td>Does not allow manipulation of values in the database</td>
</tr>
</tbody>
</table>

You can use iSQL*Plus to:

- Execute SQL statements
- Format output
- Perform calculations on data
- Print query output as reports
- Create and store script files of SQL statements for future use

Use the iSQL*Plus DESCRIBE command to display the structure of a table.

```
DESCRIBE table_name
```

```
DESCRIBE student
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>STU_ID</td>
<td>NOT NULL</td>
<td>NUMBER(9)</td>
</tr>
<tr>
<td>STU_NAME</td>
<td>NOT NULL</td>
<td>VARCHAR(25)</td>
</tr>
<tr>
<td>ADM_DATE</td>
<td></td>
<td>DATE</td>
</tr>
<tr>
<td>TUITION</td>
<td></td>
<td>NUMBER(8,2)</td>
</tr>
<tr>
<td>MAJOR_ID</td>
<td></td>
<td>NUMBER(6)</td>
</tr>
</tbody>
</table>

In the output, the Null? column indicates whether the column must contain data, and a NOT NULL value indicates that the column must contain data. The Type column displays the data type for the column.
Review Checklist: Writing Basic SQL SELECT Statements

- Identify the definitions and applications of selection, projection, and join.
- Identify SELECT statements that use the selection, projection, and join capabilities.
- Identify the purpose of a SELECT statement.
- Use arithmetic operators in SELECT statements to manipulate data.
- Modify calculations in SELECT statements to achieve the desired results.
- Know the order of operations for arithmetic operators.
- Identify the effects of parentheses in a calculation in a SELECT statement.
- Identify the proper syntax for a column alias.
- Write SELECT statements to retrieve all rows and columns from a table.
- Write SELECT statements to retrieve specific columns from a table.
- Write SELECT statements that eliminate duplicate values from the output.
- Use the concatenate operator to display literal values in the output.
- Compare and contrast SQL and iSQL*Plus.
- Identify the attributes of iSQL*Plus.
- Identify iSQL*Plus commands.
Restricting and Sorting Data
Limit the Rows Retrieved by a Query

Scope

Restrict the number of rows returned by a query by using the WHERE clause in a SELECT statement. Know the options available and the WHERE clause restrictions. Know and use the LIKE operator to restrict character values. Specify multiple conditions in the WHERE clause by making use of the logical operators. Use parentheses to override the rules of precedence. Specify criteria based on a range of values by using the BETWEEN...AND... operator. Check for null values by using the IS NULL operator.

Focused Explanation

A simple SQL SELECT statement allows you to retrieve data from the database. You can restrict the columns in your SELECT statement by listing the column names in the SELECT list. To limit the rows retrieved by a query, you can use the WHERE clause in the SELECT statement as shown in the following statement.

```
SELECT emp_id, fname, salary, joindate
FROM employee
WHERE emp_id = 3439;
```

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>FNAME</th>
<th>SALARY</th>
<th>JOINDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3439</td>
<td>John</td>
<td>5000</td>
<td>17-NOV-81</td>
</tr>
</tbody>
</table>

The WHERE clause immediately follows the FROM clause in a SELECT statement. You can specify the condition in the WHERE clause by using comparison operators. The comparison operators are:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;&gt; or !=</td>
<td>Not equal to</td>
</tr>
</tbody>
</table>
SELECT emp_id, title
FROM employee
WHERE fname = 'STEVE';

EMPLOYEE

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>TITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3388</td>
<td>ANALYST</td>
</tr>
</tbody>
</table>

The comparison operator ‘=’ can be used for character strings if you have to retrieve an exact match. In the previous example, you requested a row where the student name is STEVE by using the ‘=’ operator. Hence, in this case, you are looking for an exact match to the word ‘STEVE’. You can also restrict rows based on a character pattern. Using the LIKE operator in the WHERE clause allows you to restrict rows based on a pattern.

SELECT emp_id ID, lname NAME
FROM employee
WHERE lname LIKE '_E%';

EMPLOYEE

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>3254</td>
<td>BERNARD</td>
</tr>
</tbody>
</table>

The above statement lists the names where the second letter is ‘A’ by making use of the wildcards - _, %.

<table>
<thead>
<tr>
<th>Wildcard</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_</td>
<td>Represents any single character</td>
</tr>
<tr>
<td>%</td>
<td>Represents any number of characters</td>
</tr>
</tbody>
</table>

You cannot use column aliases in the WHERE clause. Hence, the following statement will result in an error:

SELECT emp_id, fname, (salary*12) Annsal
FROM employee
WHERE Annsal >10000;

where annsal >10000 *
ERROR at line 2:
ORA-00904: invalid column name

You should always specify the criteria for character fields and date fields by enclosing the values within single quotes.
You may have a situation wherein you have characters like ‘_’, ‘#’, and/or ‘&’ as part of the data or the character string. You cannot provide a search condition within a WHERE clause based on these values as they are reserved or special characters. In order to treat these as special characters, use the ESCAPE operator. The ESCAPE operator treats the reserved characters in a name string as literals.

```sql
SELECT id, name
FROM department
WHERE name LIKE '%\_%' ESCAPE '\';
```

**DEPARTMENT**

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1002</td>
<td>Res_Dev</td>
</tr>
<tr>
<td>1003</td>
<td>Sales_Dep</td>
</tr>
</tbody>
</table>

The backslash character escapes a single character or a symbol immediately following the backslash. The character ‘_’ is a wildcard and treated as a special character in SQL. You can escape the ‘_’ by using the '\' character as shown in the previous statement. Hence, the above SELECT statement retrieves those rows in the query result where the department name has an underscore in it.

The WHERE clause can have multiple conditions. You can specify more than one condition in the WHERE clause by using the logical operators. The logical operators are:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Returns TRUE only if both the conditions are true</td>
</tr>
<tr>
<td>OR</td>
<td>Returns TRUE if one condition is true</td>
</tr>
<tr>
<td>NOT</td>
<td>Returns TRUE if the condition is not true</td>
</tr>
</tbody>
</table>
The operators AND and OR are used to specify more than one condition in the WHERE clause.

```sql
SELECT emp_id, fname, salary, joindate
FROM employee
WHERE fname LIKE '%A%'
AND salary > 1500;
```

**EMPLOYEE**

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>FNAME</th>
<th>SALARY</th>
<th>JOINDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3166</td>
<td>PATRICK</td>
<td>2975</td>
<td>02-APR-81</td>
</tr>
<tr>
<td>3099</td>
<td>BRIAN</td>
<td>1600</td>
<td>20-FEB-81</td>
</tr>
<tr>
<td>3499</td>
<td>ADY</td>
<td>2600</td>
<td>07-NOV-81</td>
</tr>
</tbody>
</table>

The order of precedence for logical operators: NOT AND OR

```sql
SELECT emp_id, lname, dept_id, salary
FROM employee
WHERE dept_id = 1001
OR lname LIKE '%A%
AND salary >1500;
```

**EMPLOYEE**

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>LNAME</th>
<th>DEPT_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>3439</td>
<td>HOUSTON</td>
<td>1001</td>
<td>5000</td>
</tr>
<tr>
<td>3382</td>
<td>ALMEIDA</td>
<td>1001</td>
<td>2450</td>
</tr>
<tr>
<td>3388</td>
<td>CHAPELL</td>
<td>1002</td>
<td>3000</td>
</tr>
<tr>
<td>3534</td>
<td>JOHNSON</td>
<td>1001</td>
<td>1300</td>
</tr>
</tbody>
</table>

The AND operator has precedence over the OR operator. Hence, the WHERE condition operates in the following manner:

```sql
WHERE dept_id = 1001
OR (lname LIKE '%A%' AND salary >4500)
```
You can change the precedence of the logical operators by using parentheses.

```
SELECT emp_id, lname, dept_id, salary
FROM employee
WHERE (dept_id = 1001 OR lname LIKE '%A%')
AND salary > 4500;
```

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP_ID</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>3439</td>
</tr>
<tr>
<td>3382</td>
</tr>
<tr>
<td>3388</td>
</tr>
</tbody>
</table>

The criteria of selection can be based on multiple values. You cannot equate a single value with multiple values on the other side using a comparison operator. Hence, the following statement is invalid.

```
SELECT emp_id, fname
FROM employee
WHERE emp_id = 3069, 3388;
```

You can provide a list of values to be matched by using the operators IN, ANY, and ALL. You provide the list of values within parentheses.

```
SELECT emp_id, fname
FROM employee
WHERE emp_id IN (3069, 3388);
```

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP_ID</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>3388</td>
</tr>
<tr>
<td>3069</td>
</tr>
</tbody>
</table>

The IN operator functions similar to the ‘=’ comparison operator with the OR condition. The other two operators, ANY and ALL, can be used in combination with the ‘<’ and ‘>’ comparison operators.
Operators | Description
---|---
< ANY | Less than the maximum
> ANY | Greater than the minimum
< ALL | Less than the minimum
> ALL | Greater than the maximum

```
SELECT emp_id, fname, salary
FROM employee
WHERE salary > ALL (1200, 2600, 3400);
```

**EMPLOYEE**

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>FNAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>3439</td>
<td>John</td>
<td>5000</td>
</tr>
</tbody>
</table>

The above SELECT statement would retrieve those rows where salary is greater than all the values mentioned in the list. In other words, rows where the salary is greater than 3400, the maximum value in the list would be retrieved.

The search condition in the WHERE clause can be based on a range of values by making use of the BETWEEN… AND… operator.

```
SELECT emp_id, fname, salary
FROM employee
WHERE salary BETWEEN 2850 AND 5000;
```

**EMPLOYEE**

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>FNAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>3439</td>
<td>John</td>
<td>5000</td>
</tr>
<tr>
<td>3298</td>
<td>Smith</td>
<td>2850</td>
</tr>
<tr>
<td>3166</td>
<td>Patrick</td>
<td>2975</td>
</tr>
<tr>
<td>…….</td>
<td>…….</td>
<td>…….</td>
</tr>
</tbody>
</table>

The BETWEEN operator specifies the lower limit first followed by the upper limit after the AND. The query having a BETWEEN operator returns the result including the lower as well as the upper limit values.
The criteria for selection can be based on null values. The rows containing null column values can be excluded in the output by using the IS NOT NULL operator. You cannot equate any value to null because null values cannot be compared. Hence, you cannot write the expression as WHERE commission = NULL. This expression is invalid. In order to base your criteria on null values, make use of the IS NULL operator and the IS NOT NULL operator.

```
SELECT emp_id, commission
FROM employee
WHERE fname LIKE '%M%'
AND commission IS NOT NULL;
```

**EMPLOYEE**

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>COMMISSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>3444</td>
<td>0</td>
</tr>
</tbody>
</table>

This row is selected because the value in the COMMISSION column is zero, which is not the same as null.

The rows containing null column values can be shown as the output by using the IS NULL operator.

```
SELECT emp_id, commission
FROM employee
WHERE commission IS NULL
AND dept_id = 1001;
```

**EMPLOYEE**

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>COMMISSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>3439</td>
<td></td>
</tr>
<tr>
<td>3382</td>
<td></td>
</tr>
<tr>
<td>3534</td>
<td></td>
</tr>
</tbody>
</table>
Sort the Rows Retrieved by a Query

Scope

Sort the output of a query by using the ORDER BY clause. Know the syntax for sorting and sort the output in ascending or descending order.

Focused Explanation

The output of a query is shown in the order in which the rows are retrieved from the database table. There is no implicit ordering of the result. You can sort the data retrieved by a query based on column(s) or expression value(s) by using the ORDER BY clause. The ORDER BY clause sorts the rows in ascending or descending order as specified in the syntax. The default sorting option for ORDER BY clause is ascending. The ORDER BY clause should be the last clause in a SELECT statement.

```
SELECT emp_id, fname, dept_id
FROM employee
WHERE dept_id=1002
ORDER BY emp_id DESC;
```

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP_ID</td>
</tr>
<tr>
<td>3502</td>
</tr>
<tr>
<td>3476</td>
</tr>
<tr>
<td>3388</td>
</tr>
<tr>
<td>......</td>
</tr>
</tbody>
</table>

The result is sorted in descending order of EMP_ID because the keyword DESC is specified in the ORDER BY clause. Character strings are sorted alphabetically. Date values are sorted numerically and the earliest date is displayed first, if sorted in ascending order.

```
SELECT emp_id, fname name, dept_id
FROM employee
WHERE dept_id = 1001
ORDER BY name;
```

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP_ID</td>
</tr>
<tr>
<td>3534</td>
</tr>
<tr>
<td>3439</td>
</tr>
<tr>
<td>3382</td>
</tr>
<tr>
<td>......</td>
</tr>
</tbody>
</table>

Sorting can be accomplished using column aliases in the ORDER BY clause. The following statement sorts the data based on JOB_TITLE, which is an alias name for the TITLE column.

```
SELECT emp_id, fname, title job_title
FROM employee
ORDER BY job_title;
```
The ORDER BY clause can have multiple columns listed for sorting. In the example below, the result set is sorted in descending order of DEPT_ID and then in ascending order of LNAME within each DEPT_ID.

```
SELECT emp_id, lname, salary, dept_id
FROM employee
WHERE salary > 2500
ORDER BY dept_id DESC, lname;
```

EMPLOYEE

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>FNAME</th>
<th>JOB_TITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3502</td>
<td>GEORGE</td>
<td>ANALYST</td>
</tr>
<tr>
<td>3388</td>
<td>STEVE</td>
<td>ANALYST</td>
</tr>
<tr>
<td>3500</td>
<td>JAMES</td>
<td>CLERK</td>
</tr>
<tr>
<td>.......</td>
<td>.......</td>
<td>.........</td>
</tr>
</tbody>
</table>
Review Checklist: Restricting and Sorting Data

- Use the WHERE clause to restrict the rows returned by a query.
- Use the LIKE operator to search for character strings.
- Identify the use of wildcards and the ESCAPE character to treat special characters as literals.
- Use the logical operators to provide multiple WHERE conditions.
- Know the order of precedence for logical operators.
- Use parentheses to override the rules of precedence for logical operators.
- Use the IN operator to specify criteria based on a list of values.
- Use the BETWEEN….AND…. operator to specify criteria based on a range of values.
- Write SELECT statements to base your criteria on null column values by using the IS NULL operator and the IS NOT NULL operator.
- Use the ORDER BY clause to sort the output of a query.
- Use the ASC and DESC keywords to sort data in ascending and descending order respectively.
- Implement sorting on multiple columns by listing multiple columns in the ORDER BY clause.
Single-Row Functions
Describe the Various Types of Functions Available in SQL

Scope

Define single-row functions along with their attributes. Know different types of single-row and conversion functions for character, date, and numeric data.

Focused Explanation

The functions that return a single value for every row of the query are called single-row functions. There are single-row functions for number, character, and date values. You can use the single-row functions for character, number, and date manipulation as well as for conversion of date values from one data type to another. The single-row functions are:

Numeric Functions

Character Function

Date Functions

Conversion Functions

Numeric Functions

These functions take a numeric value as input and give a numeric value as output. The numeric single row functions along with their attributes are:

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUND(expr,{n})</td>
<td>Rounds off the expression to the nth place to the right of the decimal, if n is positive. If n is negative then it rounds off the expression to the nth place to the left of the decimal. n is optional. If a value for n is not specified, then expression is rounded off to the 0 decimal place.</td>
</tr>
<tr>
<td>TRUNC(expr,{n})</td>
<td>Returns the expression truncated to n decimal places. If n is not specified, then the expression is truncated to the 0 decimal place. If n is negative, the expression is truncated to the nth place to the left of the decimal.</td>
</tr>
<tr>
<td>MOD(expr, n)</td>
<td>Returns the remainder when the expression is divided by n</td>
</tr>
</tbody>
</table>
Character Functions

These functions take character values as input and generate character output values. The character single-row functions and their attributes are:

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCAT(char1, char2)</td>
<td>Concatenates char1 with char2</td>
</tr>
<tr>
<td>INITCAP(char1)</td>
<td>Capitalizes the first character of char1. If char1 consists of more than one word, the function capitalizes the first character of each word.</td>
</tr>
<tr>
<td>LOWER(char1)</td>
<td>Displays char1 in lowercase</td>
</tr>
<tr>
<td>UPPER(char1)</td>
<td>Displays char1 in uppercase</td>
</tr>
<tr>
<td>LTRIM(char1, {str})</td>
<td>Removes all the characters matching str from the left of char1. str defaults to a blank space.</td>
</tr>
<tr>
<td>TRIM({str} FROM char1)</td>
<td>Removes leading or trailing (or both) characters as specified by str from char1.</td>
</tr>
<tr>
<td>RTRIM(char1, {str})</td>
<td>Removes all the rightmost characters matching str from char1</td>
</tr>
<tr>
<td>LPAD(char1, n, {str})</td>
<td>Pads to a length of n using the characters specified by str to the left of char1. str defaults to a blank space.</td>
</tr>
<tr>
<td>RPAD(char1, n, {str})</td>
<td>Pads to a length of n using the characters as specified by str to the right of char1. str defaults to a blank space.</td>
</tr>
<tr>
<td>SUBSTR(char1, n, m)</td>
<td>Extracts m number of characters from char1 starting from the nth character</td>
</tr>
<tr>
<td>INSTR(char1, str)</td>
<td>Returns the index position of str in the string char1</td>
</tr>
<tr>
<td>LENGTH(char1)</td>
<td>Returns the length of char1</td>
</tr>
</tbody>
</table>
**Date Functions**

The single-row date functions take date fields as input. The various single-row date functions along with their attributes are:

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_MONTHS(date1, n)</td>
<td>Adds n months to date1</td>
</tr>
<tr>
<td>LAST_DAY(date1)</td>
<td>Returns the last day of the month to which date1 belongs</td>
</tr>
<tr>
<td>NEXT_DAY(date1, char1)</td>
<td>Returns the date of the first weekday as specified by char1 that falls after date1</td>
</tr>
<tr>
<td>MONTHS_BETWEEN(date1, date2)</td>
<td>Returns the number of months between date1 and date2</td>
</tr>
<tr>
<td>ROUND(date1, {fmt})</td>
<td>Returns date1 rounded to the unit as specified by fmt. fmt defaults to day.</td>
</tr>
<tr>
<td>TRUNC(date1, {fmt})</td>
<td>Returns the date1 truncated to the unit as specified by fmt</td>
</tr>
<tr>
<td>SYSDATE</td>
<td>Returns the date and time of your local database.</td>
</tr>
</tbody>
</table>

**Conversion Functions**

You can convert numeric values to character values, date values to character values, and vice-versa using conversion functions. The conversion functions are:

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TO_CHAR(date1, {fmt})</td>
<td>Converts date values into character data in the format specified by fmt</td>
</tr>
<tr>
<td>TO_CHAR(num1, {fmt})</td>
<td>Converts a number value, num1, into character data using the format specified by fmt</td>
</tr>
<tr>
<td>TO_DATE(char1, {fmt})</td>
<td>Converts a character value into date data. The format fmt is the date format of the character value char1.</td>
</tr>
<tr>
<td>TO_NUMBER(char1, {fmt})</td>
<td>Converts a character value, char1, into number data using the format specified by fmt</td>
</tr>
</tbody>
</table>
Other Single-Row functions

You have seen the character, numeric, and date single-row functions. Apart from these functions, there are a few other single row functions that are defined below:

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECODE(expr1, search1, result1, {search2, result2..} {default})</td>
<td>Returns the result if expr1 matches the search condition and returns the default if no match is found. The default is NULL.</td>
</tr>
<tr>
<td>NVL(expr1, expr2)</td>
<td>Returns expr2 if expr1 is NULL, else returns expr1</td>
</tr>
<tr>
<td>NVL2(expr1, expr2, expr3)</td>
<td>Returns expr2 if expr1 is NOT NULL, else returns expr3</td>
</tr>
<tr>
<td>NULLIF(expr1, expr2)</td>
<td>Returns NULL if expr1 is equal to expr2, else returns expr1</td>
</tr>
</tbody>
</table>
Use Character, Number, and Date Functions in SQL

Scope

Understand the usage and the syntax of the character, number, and the date functions. Convert null values to actual values using the NVL function. Use nested functions.

Focused Explanation

You can invoke single row functions from the SELECT list, WHERE clause, or HAVING clause.

Number Functions

```sql
SELECT emp_id, salary*.075, ROUND(salary*.075, 2) bonus
FROM employee
WHERE fname LIKE '%A%';
```

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP_ID</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>3166</td>
</tr>
<tr>
<td>3099</td>
</tr>
<tr>
<td>3444</td>
</tr>
<tr>
<td>......</td>
</tr>
</tbody>
</table>

The above SELECT statement rounds off the expression salary*.075 up to 2 decimal places as shown in the output above.

```sql
SELECT fname, salary, TRUNC(salary*.025, 1) comm
FROM employee
WHERE salary >= 2500;
```

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FNAME</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>JOHN</td>
</tr>
<tr>
<td>SMITH</td>
</tr>
<tr>
<td>PATRICK</td>
</tr>
<tr>
<td>GEORGE</td>
</tr>
<tr>
<td>STEVE</td>
</tr>
</tbody>
</table>

The above SELECT statement rounds off the expression salary*.025 up to 1 decimal places as shown in the output above.
The TRUNC() function can have a negative value in the second argument. In such case, this function will truncate that many digits left of the decimal point.

```
SELECT fname, salary, TRUNC(salary*.025, -1) comm
FROM employee
WHERE salary >= 2500;
```

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FNAME</td>
</tr>
<tr>
<td>JOHN</td>
</tr>
<tr>
<td>SMITH</td>
</tr>
<tr>
<td>PATRICK</td>
</tr>
<tr>
<td>GEORGE</td>
</tr>
<tr>
<td>STEVE</td>
</tr>
</tbody>
</table>

Character Functions

The following example incorporates the use of the UPPER(), LOWER() and INITCAP() functions.

```
SELECT UPPER(fname) fname, LOWER(lname) lname, INITCAP(title) title
FROM employee;
```

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FNAME</td>
</tr>
<tr>
<td>JOHN</td>
</tr>
<tr>
<td>SMITH</td>
</tr>
<tr>
<td>ROGER</td>
</tr>
<tr>
<td>.........</td>
</tr>
</tbody>
</table>

www.selftestsoftware.com
SELECT CONCAT(fname,lname) name, LPAD(salary,10,'*') new_val, INSTR(fname,'O') position, SUBSTR(lname,1,3) extract FROM employee WHERE salary >= 3000;

<table>
<thead>
<tr>
<th>NAME</th>
<th>NEW_VAL</th>
<th>POSITION</th>
<th>EXTRACT</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOHN HOUSTON</td>
<td>******5000</td>
<td>2</td>
<td>HOU</td>
</tr>
<tr>
<td>GEORGE McDonnell</td>
<td>******3000</td>
<td>3</td>
<td>McD</td>
</tr>
<tr>
<td>STEVE CHAPELL</td>
<td>******3000</td>
<td>0</td>
<td>CHA</td>
</tr>
</tbody>
</table>

The above SELECT statement concatenates the first name and last name and displays it under a single heading, NAME, pads the salary with ‘*’ on the left side, keeping the total length as 10, returns the position of the alphabet ‘O’ from FNAME, and extracts the first three characters of the last name.

You can convert null values into actual values using the NVL function.

SELECT  fname, salary, NVL(commission,1) comm FROM employee WHERE dept_id = 1003;

<table>
<thead>
<tr>
<th>FNAME</th>
<th>SALARY</th>
<th>COMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>2850</td>
<td>1</td>
</tr>
<tr>
<td>LEO</td>
<td>1250</td>
<td>1400</td>
</tr>
<tr>
<td>JAMES</td>
<td>950</td>
<td>1</td>
</tr>
<tr>
<td>.......</td>
<td>.......</td>
<td>.......</td>
</tr>
</tbody>
</table>

The above statement replaces all the null COMMISSION values with the value 1 by using the NVL function.
You can nest single-row functions. The flow of execution of nested functions is always from the inside out.

```
SELECT RPAD(TRIM(fname),10,'*') name, emp_id
FROM employee;
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>EMP_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOHN****</td>
<td>3439</td>
</tr>
<tr>
<td>SMITH*****</td>
<td>3298</td>
</tr>
<tr>
<td>ROGER*****</td>
<td>3382</td>
</tr>
</tbody>
</table>

The above SELECT statement uses the TRIM() function within the RPAD() function. Hence, the execution of the TRIM() function will occur first, and based upon its output, the RPAD function will generate the result.

**Date Functions**

The date functions operate on DATE data type values and all the date functions return DATE data type values except MONTHS_BETWEEN, which returns a numeric value. In the example below, PROB_DATE is calculated by adding three months to HIREDATE, PROJ_DATE is the first Monday that falls immediately after two months from the date of joining, and LAST_DATE is the last day of the month in which the employee was hired.

```
SELECT ADD_MONTHS(hiredate,3) PROB_DATE,
      NEXT_DAY(ADD_MONTHS(hiredate,2), 'MONDAY') PROJ_DATE,
      LAST_DAY(hiredate) LAST_DATE, SYSDATE
FROM employee
WHERE salary >= 3000;
```

<table>
<thead>
<tr>
<th>ROB_DATE</th>
<th>PROJ_DATE</th>
<th>LAST_DATE</th>
<th>SYSDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>17-FEB-82</td>
<td>23-NOV-81</td>
<td>30-NOV-81</td>
<td>08-MAY-03</td>
</tr>
<tr>
<td>03-MAR-82</td>
<td>07-DEC-81</td>
<td>31-DEC-81</td>
<td>08-MAY-03</td>
</tr>
<tr>
<td>09-MAR-83</td>
<td>13-DEC-82</td>
<td>31-DEC-82</td>
<td>08-MAY-03</td>
</tr>
</tbody>
</table>
The function `MONTHS_BETWEEN()` returns a numeric value, unlike other date functions.

```
SELECT MONTHS_BETWEEN(SYSDATE, hiredate) proj_time
FROM employee
WHERE dept_id = 1001;
```

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROJ_TIME</td>
</tr>
<tr>
<td>257.732596</td>
</tr>
<tr>
<td>262.99066</td>
</tr>
<tr>
<td>255.539047</td>
</tr>
</tbody>
</table>

`MONTHS_BETWEEN` yields a negative value if the first date falls before the second date.

```
SELECT MONTHS_BETWEEN(hiredate, SYSDATE) proj_time
FROM employee
WHERE dept_id = 1001;
```

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROJ_TIME</td>
</tr>
<tr>
<td>-257.73275</td>
</tr>
<tr>
<td>-262.99082</td>
</tr>
<tr>
<td>-255.5392</td>
</tr>
</tbody>
</table>
Use Conversion Functions

Scope

Use the TO_CHAR function to convert a number or date value to a VARCHAR2 value. Use the TO_NUMBER and TO_DATE functions to convert a character value to a number and date value respectively. Know the different options available with these conversion functions.

Focused Explanation

You can use the conversion functions to change the data type as well as the format of the input value.

TO_CHAR(number, fmt) – converts a given number into character data type in the specified format.

```sql
SELECT emp_id, TO_CHAR(salary, 'L99G999D00') amount
FROM employee;
```

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP_ID</td>
</tr>
<tr>
<td>3439</td>
</tr>
<tr>
<td>3298</td>
</tr>
<tr>
<td>3382</td>
</tr>
<tr>
<td>3166</td>
</tr>
<tr>
<td>.......</td>
</tr>
</tbody>
</table>

The format options available with TO_CHAR(number, fmt) are:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>Local currency symbol</td>
</tr>
<tr>
<td>G</td>
<td>Group separator</td>
</tr>
<tr>
<td>D</td>
<td>Decimal character</td>
</tr>
</tbody>
</table>
TO_CHAR(date,{fmt}) – converts a given date into a character string in the specified format.

```
SELECT fname, TO_CHAR(hiredate,'DDspth , day of " MONTH " in the year " YYYY") join_date 
FROM employee;
```

**EMPLOYEE**

<table>
<thead>
<tr>
<th>FNAME</th>
<th>JOIN_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOHN</td>
<td>SEVENTEENTH, day of NOVEMBER in the year 1981</td>
</tr>
<tr>
<td>SMITH</td>
<td>FIRST, day of MAY in the year 1981</td>
</tr>
<tr>
<td>ROGER</td>
<td>NINTH, day of JUNE in the year 1981</td>
</tr>
<tr>
<td>.......</td>
<td>.................</td>
</tr>
</tbody>
</table>

The formats available for the TO_CHAR(date,{fmt}) function are:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Day of the week(1-7)</td>
</tr>
<tr>
<td>Day</td>
<td>Name of the day</td>
</tr>
<tr>
<td>DD</td>
<td>Day of the month</td>
</tr>
<tr>
<td>DDD</td>
<td>Day of the year</td>
</tr>
<tr>
<td>DDspth</td>
<td>Day spelled out with a ‘th’ suffixed</td>
</tr>
<tr>
<td>Dy</td>
<td>Abbreviated name of the day</td>
</tr>
<tr>
<td>HH</td>
<td>Hour of the day(1-12)</td>
</tr>
<tr>
<td>HH24</td>
<td>Hour of the day(1-24)</td>
</tr>
<tr>
<td>MI</td>
<td>Minute(0-59)</td>
</tr>
<tr>
<td>MM</td>
<td>Month of the year(1-12)</td>
</tr>
<tr>
<td>MON</td>
<td>First three letters of the month</td>
</tr>
<tr>
<td>MONTH</td>
<td>Name of the month</td>
</tr>
<tr>
<td>AM or A.M.</td>
<td>Meridian Indicator</td>
</tr>
<tr>
<td>PM or P.M.</td>
<td>Meridian Indicator</td>
</tr>
<tr>
<td>YYYY</td>
<td>The four digits of the year</td>
</tr>
<tr>
<td>Year</td>
<td>The year spelled out</td>
</tr>
<tr>
<td>fm</td>
<td>Fill mode-removes the blank padding</td>
</tr>
</tbody>
</table>
TO_DATE(char1, fmt) – converts the given character string, which is in the format specified by the parameter fmt, to a date value.

The following example converts a character string of the format ‘DD/MM/YY’ to a date with its default format. This example uses the DUAL table. The DUAL table is a dummy table which has one column named DUMMY of VARCHAR2 data type and one row having a value ‘x’. You can use this dummy table whenever you want to select anything that is not a part of any table in the database. For example, the following SELECT statement selects a value '08/09/99', which is not a part of any table in the database.

```
SELECT TO_DATE('08/09/99', 'DD/MM/YY') date_form
FROM dual;
```

```
DUAL

<table>
<thead>
<tr>
<th>DATE_FORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>08-SEP-99</td>
</tr>
</tbody>
</table>
```

```
SELECT TO_DATE('10 of the month SEP,1999', 'DD "of the month" MON, YYYY')
FROM dual;
```

```
DUAL

<table>
<thead>
<tr>
<th>DATE_FORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>10-SEP-99</td>
</tr>
</tbody>
</table>
```
Review Checklist: Single-Row Functions

- Know the different types of single-row functions.
- Know the uses of the INSTR, NVL2, TRUNC, DECODE, TRIM, NVL, and NULLIF functions.
- Identify manipulation character functions.
- Identify the attributes of single-row functions.
- Know the syntax and use for the LPAD, INSTR, LENGTH, SUBSTR, ROUND, TRUNC, MOD, TRIM, UPPER, CONCAT, and LOWER functions.
- Use the NVL function to convert a null value to an actual value.
- Use nested functions.
- Use date and number values in calculations.
- Use the TO_CHAR function to convert a number or date value to a VARCHAR2 character string.
- Use the TO_CHAR function to format a date value.
- Use the TO_CHAR function to format a number value.
- Identify uses of the TO_DATE function.
Displaying Data from Multiple Tables
Write SELECT Statements to Access Data from More than One Table Using Equality and Nonequality Joins

Scope

Retrieve data from more than one table using equijoins and nonequijoins. Implement joins by using a join condition in the WHERE clause. Know the significance of joins by encountering a cross product or a Cartesian product. Identify the syntax for using ON, USING, and NATURAL.

Focused Explanation

A join is a condition that combines the rows of two or more tables. You provide the join condition in the WHERE clause. There are different types of joins, depending on the type of link between the tables. The join condition containing the equality operator is termed as an equijoin. In other words, the equijoin combines the rows that have equivalent values for the corresponding columns. Hence, we provide the join condition in the WHERE clause based on the matching columns. The SELECT statement followed by its output is shown below. This statement retrieves data from the EMPLOYEE table and the DEPARTMENT table by establishing an equijoin between the tables.

```
SELECT emp_id, fname, name
FROM employee, department
WHERE employee.dept_id = department.dept_id;
```

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>FNAME</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>3439</td>
<td>JOHN</td>
<td>ACCOUNTS</td>
</tr>
<tr>
<td>3298</td>
<td>SMITH</td>
<td>SALES_DEP</td>
</tr>
<tr>
<td>3382</td>
<td>ROGER</td>
<td>ACCOUNTS</td>
</tr>
<tr>
<td>......</td>
<td>.......</td>
<td>...........</td>
</tr>
</tbody>
</table>

The columns DEPT_ID of the DEPARTMENT table and DEPT_ID of the EMPLOYEE table in the above SELECT statement are prefixed with their respective table names to avoid ambiguity. You should always refer to the common columns in a query by their table names in order to avoid any confusion or ambiguity. You can give alias names to the tables and use of those alias names to prefix the column names with the table names.
In the following SELECT statement, ‘e’ and ‘d’ are the alias names given to the EMPLOYEE and DEPARTMENT tables, respectively.

```sql
SELECT emp_id, fname, name
FROM employee e, department d
WHERE e.dept_id = d.dept_id;
```

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>FNAME</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>3439</td>
<td>JOHN</td>
<td>ACCOUNTS</td>
</tr>
<tr>
<td>3298</td>
<td>SMITH</td>
<td>SALES_DEP</td>
</tr>
<tr>
<td>3382</td>
<td>ROGER</td>
<td>ACCOUNTS</td>
</tr>
</tbody>
</table>

**Cartesian Product**

The join condition becomes mandatory if you are retrieving data from more than one table. If you do not provide the join condition or provide an invalid join condition when retrieving data from more than one table, it will result in a cross product known as a Cartesian Product as shown by the statement below:

```sql
SELECT emp_id, fname, name
FROM employee, department;
```

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>FNAME</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>3439</td>
<td>JOHN</td>
<td>ACCOUNTS</td>
</tr>
<tr>
<td>3298</td>
<td>SMITH</td>
<td>ACCOUNTS</td>
</tr>
<tr>
<td>3439</td>
<td>JOHN</td>
<td>RES_DEV</td>
</tr>
<tr>
<td>3298</td>
<td>SMITH</td>
<td>RES_DEV</td>
</tr>
</tbody>
</table>

A Cartesian product is also referred to as CROSS JOIN. The output of the following statement is the same as above.

```sql
SELECT lname, fname, name
FROM employee CROSS JOIN department;
```
Natural Join

You have to specify the column name in the corresponding table in order to establish a join between the tables. This was true for 8i and earlier releases of Oracle. In Oracle9i, you have the option of allowing the Oracle server to establish the join on its own, based on the columns in the two tables, by using the NATURAL JOIN clause. To use the NATURAL JOIN clause, the column names and their data types must be the same in both the tables.

```
SELECT name, emp_id
FROM employee
NATURAL JOIN department;
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>EMP_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNTS</td>
<td>3439</td>
</tr>
<tr>
<td>SALES_DEP</td>
<td>3298</td>
</tr>
<tr>
<td>ACCOUNTS</td>
<td>3382</td>
</tr>
</tbody>
</table>

USING clause

You may have situations where more than one column name match. In such cases, NATURAL JOIN uses all the columns with matching names and data types to establish the join. You can specify the columns that should be used for creating a join condition by using the USING clause.

```
SELECT e.emp_id, e.fname, d.name
FROM employee e
JOIN department d
USING (dept_id);
```

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>FNAME</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>3439</td>
<td>JOHN</td>
<td>ACCOUNTS</td>
</tr>
<tr>
<td>3298</td>
<td>SMITH</td>
<td>SALES_DEP</td>
</tr>
<tr>
<td>.......</td>
<td>.......</td>
<td>.......</td>
</tr>
</tbody>
</table>

You are not allowed to use a table name or an alias in the USING clause.
ON clause

You can also establish an equijoin by using the ON clause. You can use the ON clause to specify column names to join, even if the column names are not the same in both the tables. The ON clause makes the code easily understandable by separating the join condition from other search conditions.

The following SELECT statement, followed by its output, joins the DEPARTMENT and EMPLOYEE tables by using the ON clause.

```sql
SELECT e.emp_id, e.fname, d.name
FROM employee e JOIN department d
ON (e.dept_id = d.dept_id)
AND fname LIKE '%H%';
```

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>FNAME</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>3439</td>
<td>JOHN</td>
<td>ACCOUNTS</td>
</tr>
<tr>
<td>3298</td>
<td>SMITH</td>
<td>SALES DEP</td>
</tr>
<tr>
<td>......</td>
<td>......</td>
<td>..........</td>
</tr>
</tbody>
</table>

In the above example, the AND condition is not the join condition between the tables. The join condition is specified in the ON clause. This makes the code easily to understand because it separates other search conditions from the join condition.

Nonequijoin

An equality operator between the matching columns of the tables establishes the equijoin condition. You can have an equijoin condition only if the values are equal in both tables. You may have a situation where the tables need to be joined but there is no relation of equality between the tables. In such cases, you can establish a nonequijoin. A nonequijoin is a condition having something other than the equality sign. The Employee table and the Grade table is an example of nonequijoin. The employees of the Employee table must have their salary between the value in the lo_sal and hi_sal columns of the Grade table. The combination of low salary and high salary have the grades assigned in the Grade table. Hence, the relation between the employee table and the grade table cannot be established by using the equality operator.
SELECT fname, salary, grd_id grade_assigned
FROM employee, grade
WHERE salary
BETWEEN lo_sal and hi_sal;

<table>
<thead>
<tr>
<th>FNAME</th>
<th>SALARY</th>
<th>GRADE_ASSIGNED</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAMES</td>
<td>950</td>
<td>1</td>
</tr>
<tr>
<td>RICHARD</td>
<td>800</td>
<td>1</td>
</tr>
<tr>
<td>RONNIE</td>
<td>1100</td>
<td>1</td>
</tr>
<tr>
<td>LEO</td>
<td>1250</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
View Data that Generally Does Not Meet a Join Condition by Using Outer Joins

Scope

View the unmatched rows in the query result, which do not appear in the equijoin, by using Outer Joins. Define the syntax of LEFT OUTER JOIN, RIGHT OUTER JOIN and FULL JOIN.

Focused Explanation

The equijoin statements retrieve only the matching records from the table. You can retrieve the unmatched records by using Outer Joins. A join between two tables that returns the matching rows as well as unmatched rows in the result is called an Outer Join. There are two types of Outer Joins, RIGHT OUTER JOINS and LEFT OUTER JOINS.

Left Outer Join

```
SELECT e.fname, d.name
FROM employee e
LEFT OUTER JOIN department d ON (e.dept_id = d.dept_id);
```

<table>
<thead>
<tr>
<th>FNAME</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOHN</td>
<td>ACCOUNTS</td>
</tr>
<tr>
<td>SMITH</td>
<td>SALES_DEP</td>
</tr>
<tr>
<td>------</td>
<td>--------------</td>
</tr>
<tr>
<td>ERICH</td>
<td>ACCOUNTS</td>
</tr>
<tr>
<td>ADY</td>
<td></td>
</tr>
</tbody>
</table>

In a Left Outer Join the unmatched rows in the table, which are present on the left side of the equality operator, are also displayed in the query result. The EMPLOYEE table’s DEPT_ID column is present on the left side of the equality operator. Hence, the query returns all the rows of the EMPLOYEE table, even if there is no match in the DEPARTMENT table. In the output shown above, the row belonging to the employee ADY does not have a corresponding value for the name of the department.

The above query is the same as:

```
SELECT e.fname, d.name
FROM employee e, department d
WHERE d.dept_id(+) = e.dept_id;
```
Right Outer Join

```
SELECT e.fname, d.name
FROM employee e
RIGHT OUTER JOIN department d ON (e.dept_id = d.dept_id);
```

<table>
<thead>
<tr>
<th>FNAME</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOHN</td>
<td>ACCOUNTS</td>
</tr>
<tr>
<td>SMITH</td>
<td>SALE_DEP</td>
</tr>
<tr>
<td>.......</td>
<td>.........</td>
</tr>
<tr>
<td>ERICH</td>
<td>ACCOUNTS</td>
</tr>
<tr>
<td>.......</td>
<td>OPERATIONS</td>
</tr>
</tbody>
</table>

In a Right Outer Join, the unmatched rows in the table, which is present on the right side of the equality operator, are also displayed in the query result. In the above SELECT statement, the Department table is on the right of the equality operator. Hence, the query returns all the rows belonging to department, even if there is no corresponding row in the EMPLOYEE table. In the output shown above, the OPERATIONS row does not have the corresponding employee information.

The above SELECT statement is similar to the Outer Join statement in earlier releases:

```
SELECT e.fname, d.name
FROM employee e, department d
WHERE d.dept_id = e.dept_id(+);
```

Full Outer Join

A Full Outer Join is a combination of a Left Outer Join and a Right Outer Join. In the case of a Full Outer Join, the extra, unmatched rows from both the tables would be displayed in the output, which was not possible in the earlier releases of Oracle.
In the example given below, the Outer Join is a Full Outer Join, which means that all the rows of the Employee table as well as all the rows of Department table would be retrieved, even if there is no matching row present in the other table. The row belonging to the employee ADY is displayed even if there is no matching value present in the Department table. Similarly, the row belonging to the department Operations is displayed, even if there is no employee working in that department.

```
SELECT e.fname, d.name
FROM employee e
FULL OUTER JOIN department d ON (e.dept_id = d.dept_id);
```

<table>
<thead>
<tr>
<th>FNAME</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOHN</td>
<td>ACCOUNTS</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ADY</td>
<td>OPERATIONS</td>
</tr>
</tbody>
</table>
Join a Table to Itself in Self-Join

Scope

Join a table to itself in a self-join. Implement the self-join using the LEFT OUTER JOIN.

Focused Explanation

You sometimes need to join the table to itself. You join a table to itself in a self-join. In a self-join, you view a table twice. An example that will help explain the concept of a self-join is that of the EMPLOYEE table which consists of information about employees and their managers as well. To retrieve the information about the employees and their respective managers, you need to view the EMPLOYEE table twice and consider the same table as EMPLOYEE table ‘e’ and MANAGER table ‘m’. You make use of the self-join and extract the information of the employee and the employee’s manager as shown below:

```
SELECT e.fname emp, m.fname manager
FROM employee e, employee m
WHERE e.mgr_id = m.emp_id;
```

<table>
<thead>
<tr>
<th>EMP</th>
<th>MANAGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>JOHN</td>
</tr>
<tr>
<td>ROGER</td>
<td>JOHN</td>
</tr>
<tr>
<td>PATRICK</td>
<td>JOHN</td>
</tr>
<tr>
<td>LEO</td>
<td>SMITH</td>
</tr>
</tbody>
</table>

Self-Join in combination with Outer Join

You have a few employees in the EMPLOYEE table who do not have any managers. You can display their information by using the self-join in combination with the Left Join Operator.

```
SELECT e.fname emp, m.fname manager
FROM employee e
LEFT OUTER JOIN employee m ON (e.mgr_id = m.emp_id);
```

<table>
<thead>
<tr>
<th>EMP</th>
<th>MANAGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>JOHN</td>
</tr>
<tr>
<td>ROGER</td>
<td>JOHN</td>
</tr>
<tr>
<td>LEO</td>
<td>SMITH</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>......</th>
<th>......</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOHN</td>
<td>......</td>
</tr>
</tbody>
</table>
Review Checklist: Displaying data from Multiple Tables

- Write SELECT statements to access data from more than one table.
- Know the syntax of Equijoin statements.
- Know the definition of Cartesian product and CROSS JOIN.
- Identify the use and syntax of the USING clause.
- Identify the use and syntax of the ON clause.
- Identify the use and syntax of NATURAL JOIN.
- Define nonequijoins and know their usage.
- View data that does not meet the join condition by using OUTER JOINS.
- Know the types of Outer Joins – Left Outer Joins, Right Outer Joins, and Full Outer Joins.
- Join a table to itself in a self-join.
Aggregating Data Using Group Functions
Identifying the Available Group Functions

Scope

Define the aggregate group functions available in SQL. Identify the aggregate functions used for character, numeric, and date columns.

Focused Explanation

You have seen single-row functions in the earlier section, which operate on a single value and generate a single value output. Group functions operate on a set of rows to generate a single output. They are mostly used with the GROUP BY clause in a SELECT statement. You can apply the aggregate functions to all the rows of a table if GROUP BY clause is not used. You can also apply the aggregate functions to a group or a set of rows if the GROUP BY clause is used.

Group Functions

The different Group functions available in SQL are:

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG(x)</td>
<td>Average value of x</td>
</tr>
<tr>
<td>MIN(x)</td>
<td>Minimum value of x</td>
</tr>
<tr>
<td>MAX(x)</td>
<td>Maximum value of x</td>
</tr>
<tr>
<td>COUNT(*)</td>
<td>Number of rows, excluding null</td>
</tr>
<tr>
<td>SUM(x)</td>
<td>Sum of x values</td>
</tr>
<tr>
<td>STDDEV(x)</td>
<td>Standard Deviation of x</td>
</tr>
<tr>
<td>VARIANCE(x)</td>
<td>Variance of x</td>
</tr>
</tbody>
</table>

You can use all the group functions with numeric values. However, you can use only the MIN(), MAX(), and COUNT() functions with character and date values.
The basic syntax for group functions is given below followed by an example that computes the maximum salary, minimum salary, the average salary and the count of the number of employees:

```
SELECT [column], group_function(column), ……
FROM table_name
[WHERE condition]
[GROUP BY column]
[ORDER BY column];
```

```
SELECT MAX(salary) maxsal, MIN(salary) minsal, AVG(salary) salavg, COUNT(*) TOTAL
FROM employee;
```

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXSAL</td>
</tr>
<tr>
<td>5000</td>
</tr>
</tbody>
</table>

All the group functions consider duplicate values. The following SELECT statement computes the count of DEPT_ID in the EMPLOYEE table. More than one employee may be working in a particular department. Hence, the DEPT_ID column of the EMPLOYEE table will have duplicate values.

```
SELECT COUNT(dept_id) no_of_dept
FROM employee;
```

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_OF_DEPT</td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>

You can use the DISTINCT clause within the group function to eliminate duplicate values as shown by the following SELECT statement:

```
SELECT COUNT(DISTINCT(dept_id)) no_of_dept
FROM employee;
```

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_OF_DEPT</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
Use Group Functions

Scope

Use the Group Functions and force the group functions to include null values. Identify the use of the aggregate functions and the errors in the group functions WHERE clause.

Focused Explanation

All the group functions ignore null values. However, you can force the group functions to include null values by making use of the NVL function within the group function. The statement below calculates the average commission as the total commission of all the employees divided by the number of employees who receive commission.

```
SELECT AVG(commission) 
FROM employee;
```

```
EMPLOYEE

<table>
<thead>
<tr>
<th>AVG(COMMISSION)</th>
</tr>
</thead>
<tbody>
<tr>
<td>550</td>
</tr>
</tbody>
</table>
```

You can use NVL to include null values.

```
SELECT AVG(NVL(commission,0)) comm_avg 
FROM employee;
```

```
EMPLOYEE

<table>
<thead>
<tr>
<th>COMM_AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>157.142857</td>
</tr>
</tbody>
</table>
```

You cannot use the WHERE clause to restrict groups. In order to restrict groups, you use the HAVING clause.

```
SELECT AVG(salary)  
FROM employee      
WHERE AVG(salary) > 2000  
GROUP BY dept_id;
```

The above statement results in the following error:

```
SELECT AVG(salary) FROM employee WHERE AVG(salary) > 2000 GROUP BY dept_id;  
ERROR at line 1:  
ORA-00934: group function is not allowed here
```

Hence, you cannot restrict a group function by using a WHERE clause.
The group functions can take column names, expressions, constants, or functions as parameters. A few examples are given below, along with their output, implementing group functions:

```
SELECT MIN(salary), MAX(salary)
FROM employee;
```

<table>
<thead>
<tr>
<th>MIN(SALARY)</th>
<th>MAX(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>5000</td>
</tr>
</tbody>
</table>

```
SELECT SUM(salary*0.5) total
FROM employee;
```

<table>
<thead>
<tr>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>15812.5</td>
</tr>
</tbody>
</table>

```
SELECT AVG(NVL(commission,0)) comm
FROM employee;
```

<table>
<thead>
<tr>
<th>COMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>146.666667</td>
</tr>
</tbody>
</table>

```
SELECT COUNT(*) TOTAL
FROM employee;
```

<table>
<thead>
<tr>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
</tr>
</tbody>
</table>

COUNT(*) yields the same output as the following SELECT statement where the PRIMARY KEY column is used with the COUNT() function.

```sql
SELECT COUNT(emp_id)
FROM employee;
```

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL</td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>
Group Data Using GROUP BY Clause

Scope

Group data and split the entire table into small groups by using the GROUP BY clause and also learn the syntax to use the same. Encounter the errors and identify the points to be remembered while using the GROUP BY clause.

Focused Explanation

The group functions you have used so far operate on the entire table as a single group. You can further divide the table into small groups and apply these group functions on those groups by using the GROUP BY clause. The GROUP BY clause allows you to divide the table into smaller groups. The GROUP BY clause follows the WHERE or the HAVING clause in a SELECT statement.

GROUP BY clause

```
SELECT dept_id, AVG(salary)
FROM employee
GROUP BY dept_id;
```

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>AVG(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>2916.66667</td>
</tr>
</tbody>
</table>

You must include all those column names in the GROUP BY clause that you have listed in your SELECT list other than the group functions; else you will encounter the following error:

```
SELECT dept_id, mgr_id, AVG(salary)
FROM employee
GROUP BY dept_id;
```

```
select dept_id, mgr_id, avg(salary) from employee
* 
ERROR at line 1:
ORA-00979: not a GROUP BY expression
```

Hence, the correct statement would be:

```
SELECT dept_id, mgr_id, AVG(salary)
FROM employee
GROUP BY dept_id, mgr_id;
```
### EMPLOYEE

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>MGR_ID</th>
<th>AVG(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>3382</td>
<td>1300</td>
</tr>
<tr>
<td>1001</td>
<td>3439</td>
<td>2450</td>
</tr>
<tr>
<td>..........</td>
<td>..........</td>
<td>..........</td>
</tr>
</tbody>
</table>
Include or Exclude Grouped Rows by Using the HAVING Clause

Scope

Include or exclude the groups by using the HAVING clause and understand how it is different from the WHERE clause.

Focused Explanation

You can restrict the group results just as you restrict the rows in the query result. You use the WHERE clause to restrict the rows in the query result, whereas to restrict group results, you use the HAVING clause. The HAVING clause works in the same manner as the WHERE clause, but for group functions. The following SELECT statement shows only those rows where the average salary is greater than 2000.

```
SELECT dept_id, AVG(salary)
FROM employee
HAVING AVG(salary) > 2000
GROUP BY dept_id;
```

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>AVG(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>2916.66667</td>
</tr>
</tbody>
</table>

You use the WHERE clause to restrict the rows and the HAVING clause to restrict the groups. You can combine both the WHERE clause and the HAVING clause in a SELECT statement.

```
SELECT dept_id, SUM(salary)
FROM employee
WHERE title NOT LIKE '%SALES%'
GROUP BY dept_id
HAVING SUM(salary) >7500
ORDER BY SUM(salary);
```

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>8750</td>
</tr>
<tr>
<td>1002</td>
<td>10875</td>
</tr>
</tbody>
</table>
You can use the logical operators with the HAVING clause as you do in case of the WHERE clause to include multiple restrictions.

```sql
SELECT dept_id, MIN(salary), MAX(salary)
FROM employee
WHERE title NOT LIKE '%SALES%'
GROUP BY dept_id
HAVING MIN(salary) < 1500 AND MAX(salary) > 3000;
```

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>MIN(SALARY)</th>
<th>MAX(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>1300</td>
<td>5000</td>
</tr>
</tbody>
</table>

The above example returns those rows where the minimum salary is less than 1500 and the maximum salary is more than 3000.
Review Checklist: Aggregating Data Using Group Functions

- Identify the group functions available in SQL.
- Use the AVG(), SUM(), MAX(), MIN(), COUNT() group functions and know their significance.
- Identify syntax errors in group functions WHERE clause.
- Pass columns, functions, expressions as parameters to group functions.
- Group data using the GROUP BY clause.
- Include or exclude the groups in the result set by using the HAVING clause.
Subqueries
Describe the Types of Problems that Subqueries Can Solve

Scope

Identify the output that would require a subquery or join. Define the purpose and the execution of a subquery. Understand the working of the outer query to generate output based on the result generated by the inner query.

Focused Explanation

In earlier sections, you have seen a scenario where you established a join to return the desired output. For example, you need to retrieve records of employees who earn more than the salary of employee ADY. You can retrieve the desired result by using the self-join, considering the employee table as the e table, which will contain records of all employees and the a table, which will contain ADY’s records, as follows:

```
SELECT e.fname emp_name, e.salary emp_sal,
FROM employee e, employee a
WHERE e.salary > a.salary AND a.fname = ‘ADY’;
```

The above statement will display the following output:

<table>
<thead>
<tr>
<th>EMP_NAME</th>
<th>EMP_SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOHN</td>
<td>5000</td>
</tr>
<tr>
<td>SMITH</td>
<td>2850</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

You can achieve the above output using subqueries. A subquery is a query placed within another query. You first need to know ADY’s salary, with which you can compare the salaries of other employees. You can have one query or a SELECT statement to retrieve ADY’s salary and another to identify the employee who earns more than that amount, as shown below:

```
SELECT fname, salary
FROM employee
WHERE salary > (SELECT salary
FROM employee
WHERE fname = ‘ADY’);
```

A subquery is always enclosed within parentheses. The value returned by the inner query must be of the same data type as the column of the outer query with which it is being compared. You can use a subquery to select rows from the table based on an unknown condition. In the above example, you are retrieving rows of employees who earn more than ADY. In this case, you do not know ADY’s salary.
Therefore, the search condition is unknown, which would be retrieved by the inner query. Apart from using subqueries in a SELECT statement, you can also use subqueries in INSERT, UPDATE, and DELETE statements. The following example uses a subquery in the INSERT statement:

```
INSERT INTO employee_copy
SELECT *
FROM employee
WHERE emp_id = 3069;
```

This inserts the row that belongs to the employee whose employee ID is equal to 3069 into the EMPLOYEE_COPY table.

The next example uses a subquery to update the EMPLOYEE_COPY table’s DEPT_ID column with a value equal to that of the department number of employee 3121 in the EMPLOYEE table.

```
UPDATE employee_copy
SET dept_id = (SELECT deptno
               FROM employee
               WHERE emp_id = 3121);
```

You can also use a subquery in DELETE statements, as follows:

```
DELETE FROM employee_copy
WHERE dept_id = (SELECT id
                 FROM department
                 WHERE name = 'ACCOUNTS');
```

The above example deletes all rows from the EMPLOYEE_COPY table, where the department number is equal to that of the ACCOUNTS department.
Using a subquery is the same as performing two queries in sequence and using the result of the first query in the second query. The inner query executes once for each execution of the main query. The subquery is also known as a nested query. The subquery executes first, and its output is used to complete the query condition for the main or outer query. However, the inner query does not have to always return a value to the outer query. For example, the following SELECT statement returns “no rows selected” because the inner query does not return any row. This is because there is no employee with employee ID equal to 1000.

```
SELECT * FROM department
WHERE id = (SELECT dept_id
FROM employee
WHERE emp_id = 1000);
```

There are certain guidelines that must be followed while defining subqueries. The guidelines are:

- Always enclose a subquery in parentheses.
- Place subqueries to the right of the comparison operator.
- The ORDER BY clause is not required in the inner query or the subquery.
- Use single-row operators with single-row subqueries and multiple-row operators with multiple-row subqueries.
Define Subqueries and Write Single-Row and Multiple-Row Subqueries

Scope

Identify the characteristics of subqueries in terms of the number of rows returned by the subquery and the number of subqueries the main query can have. Define a single-row subquery and a multiple-row subquery based on the number of rows returned by the subquery. Use statements in which the main query and the subquery retrieve data from different tables. Implement SELECT statements where multiple columns or expressions can be compared between the main query and the subquery. Identify the results of using the equality operator in the outer SQL statement in single-row and multiple-row subqueries. Identify the operators that can be used with single-row and multiple-row subqueries.

Focused Explanation

A subquery is a SELECT statement that can be embedded in a clause of another SELECT statement. The subquery returns a value to the main query based on which the main query generates the final output. A subquery can return multiple rows to the main query. Subqueries are of two types based on the number of rows returned by them.

Single-Row Subqueries

A subquery that returns only one row to the main query is called a single-row subquery. For example, the following subquery returns only one row, the salary of the employee GEORGE, to the main query.

```
SELECT emp_id, fname
FROM employee
WHERE salary = (SELECT salary
                FROM employee
                WHERE fname = 'GEORGE');
```

Multiple-Row Subqueries

A subquery that returns more than one row is called a multiple-row subquery. The following subquery is an example of a multiple-row subquery because there are multiple employees in department 1001. As a result, the inner query returns more than one row to the outer query. In case of a multiple-row subquery, you use a multiple-row operator, such as IN, ANY, and ALL.

```
SELECT emp_id, dept_id, salary, mgr_id
FROM employee
WHERE mgr_id IN (SELECT emp_id
                  FROM employee
                  WHERE dept_id= 1001);
```

The main query can have multiple subqueries. A subquery can contain another subquery. You can nest up to 255 levels of subqueries in the WHERE clause.
You can have the subquery and the main query retrieve data from different tables. The following example has the subquery retrieving data from the DEPARTMENT table and the main query retrieving data from the EMPLOYEE table.

```
SELECT emp_id, fname, dept_id
FROM employee
WHERE dept_id = (SELECT id
                 FROM department
                 WHERE name = 'ACCOUNTS');
```

Single-row subqueries can return only one row but multiple columns. On the other hand, multiple-row subqueries can return multiple rows and multiple columns. You can compare multiple columns between the subquery and the main query.

```
SELECT emp_id, salary, dept_id
FROM employee
WHERE (salary, dept_id) = (SELECT salary, dept_id
                            FROM employee
                            WHERE fname = 'GEORGE');
```

In the above example, the main query compares two columns, SALARY and DEPT_ID, with the subquery. In the example, the subquery returns only one row. As a result, you can use the equality operator for comparison. However, you must use the multiple-row operator if the subquery returns multiple rows. In this case, the multiple-row comparison operator, IN, replaces the single-row equality operator. In the following example, the subquery returns multiple rows. Hence, you use the IN operator.

```
SELECT emp_id, salary, dept_id
FROM employee
WHERE (salary, dept_id) IN (SELECT salary, dept_id
                            FROM employee
                            WHERE title = 'MANAGER');
```

Single-row subqueries can use single-row comparison operators as well as multiple-row comparison operators. On the other hand, a multiple-row subquery can use only multiple-row comparison operators; else, you will encounter the following error:

```
SELECT emp_id, salary, dept_id
FROM employee
WHERE (salary, dept_id) = (SELECT salary, dept_id
                            FROM employee
                            WHERE title = 'MANAGER');

WHERE (salary, dept_id) = (SELECT salary, dept_id
                            FROM employee
                            WHERE title = 'MANAGER')
ERROR at line 2:
ORA-01427: single-row subquery returns more than one row
```
A single-row subquery fails if it returns multiple rows. However, by changing the comparison operator to a multiple-row comparison operator, you can execute the above statement in the following manner:

```sql
SELECT emp_id, salary, dept_id
FROM employee
WHERE (salary, dept_id) IN (SELECT salary, dept_id
                             FROM employee
                             WHERE title= 'MANAGER');
```

Single-row comparison operators, such as =, >, <, >=, <=, and <>, can be used with single-row subqueries. The following example retrieves information about employees who earn more than or equal to the employee, SMITH.

```sql
SELECT emp_id, fname, salary
FROM employee
WHERE salary >= (SELECT salary
                 FROM employee
                 WHERE fname = 'SMITH');
```

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP_ID</td>
</tr>
<tr>
<td>3439</td>
</tr>
<tr>
<td>....</td>
</tr>
</tbody>
</table>

You cannot use single-row comparison operators in multiple-row subqueries. You use multiple-row comparison operators, such as the IN, NOT IN, ALL, and ANY operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>Exactly equal to any value in the list</td>
</tr>
<tr>
<td>NOT IN</td>
<td>Not equal to any value in the list</td>
</tr>
<tr>
<td>&gt;ANY</td>
<td>Greater than at least one value in the list</td>
</tr>
<tr>
<td>&lt;ANY</td>
<td>Less than at least one value in the list</td>
</tr>
<tr>
<td>&gt;ALL</td>
<td>Greater than all values in the list</td>
</tr>
<tr>
<td>&lt;ALL</td>
<td>Less than all values in the list</td>
</tr>
</tbody>
</table>

```sql
SELECT salary, emp_id
FROM employee
WHERE salary >ANY (SELECT salary
                    FROM employee
                    WHERE dept_id = 1001);
```

In the above example, the subquery returns three values, 5000, 2450, and 1300. Therefore, the above statement is the same as:

```sql
SELECT salary, emp_id
FROM employee
WHERE salary >ANY (5000, 2450,1300);
```
The >ANY operator returns rows in which salary is greater than the minimum value of the list, which is 1300. As a result, the output will be:

<table>
<thead>
<tr>
<th>SALARY</th>
<th>EMP_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>3439</td>
</tr>
<tr>
<td>2850</td>
<td>3298</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The multiple-row comparison operator, >ALL, returns rows in which the value is greater than the maximum value in the list. For example,

```sql
SELECT emp_id, fname, salary
FROM employee
WHERE salary >ALL (SELECT salary
                  FROM employee
                  WHERE title = 'CLERK');
```

In the above example, the subquery returns four values, 950, 800, 1100, and 1300. Therefore, the above statement is equivalent to the following statement:

```sql
SELECT emp_id, fname, salary
FROM employee
WHERE salary >ALL (950,800,1100,1300);
```

The >ALL operator returns rows in which the salary is greater than the maximum value of the list, which is 1300. As a result, the output will be:

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>FNAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>3439</td>
<td>JOHN</td>
<td>5000</td>
</tr>
<tr>
<td>3298</td>
<td>SMITH</td>
<td>2850</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
You have seen that the main query generates output based on the values returned by the subquery. You may have situations where the main query does not return any data but the inner query returns a result. For example, you need to retrieve information about employees in the Operations department. You have a subquery that returns the department id of the Operations department to the main query and, based on that value, the main query returns the output.

```sql
SELECT emp_id, fname, salary, dept_id
FROM employee
WHERE dept_id = (SELECT dept_id
                 FROM department
                 WHERE name = 'OPERATIONS');
```

The final output of the above query is "no rows selected". This query does not return any data because there is no employee working in the OPERATIONS department. In this example, the inner query or the subquery returns data, which is the department number of the OPERATIONS department, but the main query does not return any result. However, if the inner query or subquery returns a null value, the main query will also return a null value. For example,

```sql
SELECT fname, dept_id, salary, commission
FROM employee
WHERE commission = (SELECT commission
                     FROM employee
                     WHERE fname = 'JOHN');
```

The above query results in a null value because the inner query returns a null value. The inner query returns null because JOHN earns null commission. The NOT IN operator results in a null value if the inner query returns null as one of the values in the result set. In the following example, the SELECT statement attempts to retrieve information about employees who do not have any subordinates.

```sql
SELECT fname, mgr_id
FROM employee
WHERE emp_id NOT IN (SELECT mgr_id
                      FROM employee);
```

Logically, the main query should return 12 rows in the result. However, it does not return any rows because one of the values returned by the inner query is null. Hence, do not use the NOT IN operator when null values are likely to be part of the result set.
Using Group Functions in a Subquery

You can have a subquery in a SELECT statement that uses group functions as shown below:

```
SELECT dept_id, AVG(salary)
FROM employee
GROUP BY dept_id
HAVING AVG(salary) >ANY (SELECT AVG(salary)
                          FROM employee);
```

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>AVG (SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>2916.66667</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The above example displays the department number along with the average salary of that department where the average salary is more than the average salary of department 1002. You can use subqueries not only in the WHERE clause but also in the HAVING clause.
List the Types of Subqueries

Scope

Identify the clauses of an SQL statement that can contain subqueries. Use a subquery in the FROM and WHERE clauses of a SELECT statement and in the UPDATE, DELETE, and INSERT statements.

Focused Explanation

You can use a subquery in different clauses of a SELECT statement, an UPDATE statement, a DELETE statement, and an INSERT statement.

A subquery in the FROM clause of a SELECT statement

You can use a subquery in the FROM clause of a SELECT statement. The subquery used in the FROM clause of a SELECT statement is known as an inline view. You specify an alias name to the FROM clause subquery and refer to its column by its alias name. You can use the subquery in the FROM clause, as follows:

```sql
SELECT e.emp_id, e.fname, e.salary, a.dept_id, a.salavg
FROM employee e, (SELECT dept_id, AVG(salary) salavg
                   FROM employee
                   GROUP BY dept_id) a
WHERE e.dept_id = a.dept_id
AND e.salary > a.salavg;
```

The above SELECT statement retrieves rows in which employees earn more than the average salary of their departments. The subquery in the FROM clause has two columns, DEPT_ID and SALAVG, and the values are:

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>SALAVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>2916.66667</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

In the above example, the FROM clause subquery ‘a’ returns details of all department numbers and the average salary from the Employee table. The WHERE e.dept_id = a.dept_id AND e.salary > a.salavg clause of the main query displays the employee id, the first name, the salary, the department number, and the average salary of the department for all employees who earn more than the average salary of their departments.
A subquery in the WHERE clause of a SELECT statement

You have seen, in the earlier section, how to use the subquery in the WHERE clause of the SELECT statement. The following example uses a subquery in the WHERE clause:

```sql
SELECT emp_id, fname, salary
FROM employee
WHERE dept_id = (SELECT id
FROM department
WHERE name = 'SALES_DEP');
```

The subquery returns 1003 as the value to the main query. As a result, the above statement becomes:

```sql
SELECT emp_id, fname, salary
FROM employee
WHERE dept_id = 1003;
```

A subquery in the SET clause of an UPDATE statement

You can use subqueries in the SET clause of an UPDATE statement. The following example shows how to use a subquery in the SET clause of an UPDATE statement:

```sql
UPDATE employee_copy
SET (salary, commission) = (SELECT salary, commission
FROM employee
WHERE emp_id = 3388)
WHERE dept_id = 1002;
```

The above example will update the SALARY and COMMISSION columns of the employees working in department 1002 with a value equal to the salary and commission of employee 3388. The salary and commission of employee 3388 is retrieved by the subquery in the SET clause.

A subquery in the INTO clause of an INSERT statement

In earlier sections, you have seen how to use a subquery in an INSERT statement, as follows:

```sql
INSERT INTO employee_copy
SELECT *
FROM employee
WHERE dept_id = 1001;
```

Similarly, you can use the subquery in the INTO clause of the INSERT statement. For example,

```sql
INSERT INTO (SELECT emp_id, fname, salary
FROM employee_copy)
VALUES (3107,"RONALD",3580);
```

The above statement inserts value into three columns, EMP_ID, FNAME, and SALARY, as listed in the subquery. You can use a subquery in the INTO clause of an INSERT statement but not in the VALUES clause of the INSERT statement. You cannot use the VALUES keyword when you use a subquery to insert values in the table. You directly write the subquery instead of the VALUES clause.
A subquery in the WHERE clause of a DELETE statement

You can also use subqueries in DELETE statements. You can use a subquery in the WHERE clause of the DELETE statement, as in the following example:

```
DELETE FROM employee_copy
WHERE dept_id = (SELECT dept_id
                 FROM employee
                 WHERE empno = 3388);
```
Review Checklist: Subqueries

- Identify the significance of subqueries.
- Understand how subqueries work and how the main query generates output based on the inner query.
- Define guidelines to use subqueries.
- Understand the characteristics of the subquery in terms of the number of rows returned by the subquery and the number of subqueries the main query can have.
- Differentiate between a single-row subquery and a multiple-row subquery.
- Retrieve data in the main query and the subquery from different tables.
- Compare multiple columns between the main query and the subquery.
- Use single-row comparison operators with single-row subqueries and multiple-row comparison operators with multiple-row subqueries.
- Use a subquery in different clauses of the SELECT statement.
- Use a subquery in the INSERT, UPDATE, and DELETE statements.
Producing Readable Output with iSQL*Plus
Produce Queries that Require a Substitution Variable

Scope

Use the single-ampersand (&) substitution variable to accept column values and table and column names as user input. Define the purpose of the double ampersand (&&) substitution variable. Identify the uses of substitution variables.

Focused Explanation

You have used the fixed WHERE clause and the fixed SELECT list. Using the iSQL*Plus, you can create reports or scripts that prompt users to supply their own values to restrict the range of data returned by substitution variables. A substitution variable is a variable that substitutes the value provided by the user when the statement is executed. A substitution variable acts as a container in which you can temporarily store values. The iSQL*Plus substitution variable makes the query interact with the user.

The ‘&’ Substitution Variable

In iSQL*Plus, you can use the single ampersand ‘&’ substitution variable to prompt the user for values and temporarily store the values in it. You may often need to dynamically restrict the returned data. You can do this by using a variable prefixed with an ampersand ('&') to prompt the user for a value. The following example uses a substitution variable, d, prefixed with an ampersand to prompt the user for a value for DEPT_ID.

```
SELECT emp_id, fname, salary
FROM employee
WHERE dept_id = &deptid;
```

The above statement creates a substitution variable, DEPTID, which prompts for a value in the following manner:

```
Enter value for deptid: 1003
```

You enter value 1003 for the variable ‘DEPTID’ and the statement becomes equivalent to:

```
SELECT emp_id, fname, salary
FROM employee
WHERE dept_id = 1003;
```

This statement retrieves the records that belong to employees who work in department 1003.

The value, 1003, is stored in the variable 'deptid' only until you execute the statement. The next time you execute the same SELECT statement, you would be again prompted to enter a value for the variable 'deptid'. Therefore, the single-ampersand (&) substitution variable allows you to enter a new value each time the statement is executed. When iSQL*Plus encounters a single-ampersand(&), you are prompted to enter a value for the substitution variable named in the SQL statement. You must enclose the substitution variable within quotes while using it for character or date values because character and date values are always enclosed within single quotation marks.
The following example shows how to use substitution variables for character values:

```
SELECT *
FROM employee
WHERE fname = '&fn';
Enter value for fn: SMITH
```

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP_ID</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>3298</td>
</tr>
</tbody>
</table>

You can supplement the following using substitution variables:

- WHERE condition
- ORDER BY clauses
- Column expressions
- Table names
- Entire SELECT statement

Supplementing column names, expressions, and text

You can use a substitution variable not only in a WHERE condition but also to substitute column names or expressions. The following example uses substitution variables to substitute column names, expressions, and the ORDER BY clause:

```
SELECT emp_id, fname, &col_name
FROM employee
WHERE &condition
ORDER BY &order_col;
```

You would be prompted to enter values for substitution variables when you execute the above statement.

Enter value for col_name: salary
Enter value for condition: salary > 2000
Enter value for order_col: salary desc

After you have provided values for the substitution variables, the above statement will be equivalent to:

```
SELECT emp_id, fname, salary
FROM employee
WHERE salary > 2000
ORDER BY salary desc;
```
Supplementing the Entire SELECT Statement

You can substitute the entire SELECT statement using the single-ampersand (&) substitution variable. For example, the following statement uses the single-ampersand substitution variable to substitute the entire SELECT statement:

```
SELECT &statement;
```

Enter value for statement: dept_id, name FROM department

The above statement would retrieve the department id and the name of the department from the DEPARTMENT table. The output will be:

```
DEPARTMENT

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>ACCOUNTS</td>
</tr>
<tr>
<td>1002</td>
<td>RES_DEV</td>
</tr>
<tr>
<td>.....</td>
<td>.....</td>
</tr>
</tbody>
</table>
```

The ‘&&’ Substitution Variable

You can use the double-ampersand substitution variable if you need to retain the value of the variable so that the user is not prompted for the value each time the variable is being used. The user will be prompted only once for the value of the variable. The following example defines a double-ampersand substitution variable, which will prompt the user for a value the first time the statement is executed.

```
SELECT emp_id, fname, title
FROM employee
WHERE title = ‘&&title’;
```

Enter value for title: MANAGER

```
EMPLOYEE

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>FNAME</th>
<th>TITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3298</td>
<td>SMITH</td>
<td>MANAGER</td>
</tr>
<tr>
<td>3382</td>
<td>ROGER</td>
<td>MANAGER</td>
</tr>
<tr>
<td>.....</td>
<td>.....</td>
<td>.....</td>
</tr>
</tbody>
</table>
```
The value ‘MANAGER’ would be stored in the substitution variable ‘TITLE’ until you explicitly undefine it or exit the iSQL*Plus session. The command to undefine the substitution variable is:

```
SQL> UNDEFINE title
```

The UNDEFINE command is an iSQL*Plus command to remove the value from substitution variables, executed at the SQL prompt.
Produce Readable Output

Scope

Use the iSQL*Plus COLUMN command to control the display of a column. Define the various options available with the FORMAT option of the COLUMN command.

Focused Explanation

You see database column names as headings for columns in the result of the SELECT statement. There is no formatting being performed on columns and headings and these appear in uppercase unless you specify the casing of the heading within double quotes in the SELECT statement. The iSQL*Plus command allows you to create column headings and format column values using the iSQL*Plus COLUMN command. This command controls the display of the column. The general syntax for the COLUMN command is:

```
COLUMN [{column | alias} [option]]
```

The COLUMN command options available are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLE[AR]</td>
<td>Clears the column format</td>
</tr>
<tr>
<td>HEA[DING]</td>
<td>Sets text as the heading for the column (a vertical bar(</td>
</tr>
<tr>
<td>FOR[MAT]</td>
<td>Formats column data by changing its display</td>
</tr>
<tr>
<td>NOPRI[NT]</td>
<td>Prevents the column from being displayed</td>
</tr>
<tr>
<td>PRI[NT]</td>
<td>Displays the column</td>
</tr>
<tr>
<td>NUL[L]</td>
<td>Specifies the text to be displayed for null values</td>
</tr>
</tbody>
</table>

You can use the COLUMN command to format columns before issuing a SELECT statement. For example, you issue the following SELECT statement:

```
SELECT fname, salary, commission, dept_id FROM employee;
```
The output of the previous statement, which would be displayed without formatting, is:

<table>
<thead>
<tr>
<th>FNAME</th>
<th>SALARY</th>
<th>COMMISSION</th>
<th>DEPT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOHN</td>
<td>5000</td>
<td>1001</td>
<td></td>
</tr>
<tr>
<td>SMITH</td>
<td>2850</td>
<td>1003</td>
<td></td>
</tr>
<tr>
<td>ROGER</td>
<td>2450</td>
<td>1001</td>
<td></td>
</tr>
</tbody>
</table>

Database column names appear as column headings in the output. However, you can change the display of columns using the COLUMN command before the SELECT statement, as follows:

```
SQL> COLUMN fname HEADING 'Employee | Name'
SQL> COLUMN salary JUSTIFY LEFT FORMAT $9,999.00
SQL> COLUMN commission FORMAT 99999999999 NULL 'No Commission'
```

The FORMAT option for the COLUMN command has the following models:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Represents a number</td>
</tr>
<tr>
<td>0</td>
<td>Forces the display of a zero</td>
</tr>
<tr>
<td>$</td>
<td>Represents the floating dollar sign</td>
</tr>
<tr>
<td>L</td>
<td>Represents local currency</td>
</tr>
<tr>
<td>.</td>
<td>Specifies the position of a decimal point</td>
</tr>
<tr>
<td>,</td>
<td>Is the thousand separator</td>
</tr>
</tbody>
</table>

The above COLUMN command sets Employee Name as the heading of the FNAME column in the SELECT list, the SALARY column is left justified and displayed in the format $9,999.00, and the COMMISSION column displays No Commission for rows where commission is null. As a result, the output of the SELECT statement changes to the following:
EMPLOYEE

<table>
<thead>
<tr>
<th>Employee Name</th>
<th>Salary</th>
<th>Commission</th>
<th>DEPT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOHN</td>
<td>$5,000.00</td>
<td>No Commission</td>
<td>1001</td>
</tr>
<tr>
<td>SMITH</td>
<td>$2,850.00</td>
<td>No Commission</td>
<td>1003</td>
</tr>
</tbody>
</table>

The settings made for columns using the COLUMN command will last until you exit iSQL*Plus or issue the following command to explicitly clear settings:

SQL> COLUMN fname CLEAR

You can display the current settings of a column by executing the following command:

SQL> COLUMN salary

You can clear the settings of all columns in a single step by issuing the following command:

SQL> CLEAR COLUMN

You can suppress duplicate values by using the iSQL*Plus BREAK command, which divides rows into sections and suppresses duplicate values. The BREAK command would be effective if you have used the ORDER BY clause in your SELECT statement. You can set a break on a particular column, as follows:

SQL> BREAK ON dept_id
Now you may execute the following statement:

```
SELECT dept_id, fname, salary
FROM employee
ORDER BY dept_id;
```

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>FNAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>JOHN</td>
<td>5000</td>
</tr>
<tr>
<td></td>
<td>ROGER</td>
<td>2450</td>
</tr>
<tr>
<td></td>
<td>ERICH</td>
<td>1300</td>
</tr>
<tr>
<td>1002</td>
<td>PATRICK</td>
<td>2975</td>
</tr>
<tr>
<td></td>
<td>STEVE</td>
<td>3000</td>
</tr>
</tbody>
</table>

The BREAK command does not repeat values in the column on which the break has been set.

The TTITLE and BTITLE commands set the header and footer for a report.

You can set the report header using the following command:

```
SQL> TTITLE 'Employee | Report'
```

You can set the report footer using the following command:

```
SQL> BTITLE 'End of Report'
```

The following table describes a few SET command variables:

<table>
<thead>
<tr>
<th>SET command variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECHO</td>
<td>Lists commands in a script file when set to ON</td>
</tr>
<tr>
<td>VERIFY</td>
<td>Confirms changes in the SQL statement if set to ON</td>
</tr>
<tr>
<td>PAGESIZE(24[n])</td>
<td>Sets the number of lines on each page. The default value is 24</td>
</tr>
<tr>
<td>LINESIZE(80[n])</td>
<td>Sets the number of lines on each page. The default value is 80.</td>
</tr>
<tr>
<td>FEEDBACK(6[n])</td>
<td>Displays the number of records returned by a query when the query selects at least n records. The default value is 6.</td>
</tr>
<tr>
<td>ARRAYSIZE(20[n])</td>
<td>Sets the database data fetch size</td>
</tr>
<tr>
<td>TERMOUT</td>
<td>Controls the display of the output of commands executed from a script file</td>
</tr>
</tbody>
</table>
Create and Execute Script Files

Scope

Create a script file and run an iSQL*Plus report. List the steps to save a text file as a script file and execute the script.

Focused Explanation

You have been entering iSQL*Plus commands at the SQL prompt. You can place all commands, including SELECT statements, in a command or a script file and execute the script file. A typical script consists of at least one SQL statement along with several iSQL*Plus commands.

Creating a Script File to Run a Report

To create a script file to run a report:

1. Create the SQL statement at the SQL prompt. Test the statement for its correct execution. Ensure that the ORDER BY clause is included if you intend to use breaks.

2. Open a script file by typing the following command at the SQL prompt and press Enter:

   SQL> ED script_file

   This creates and opens a .sql file by the name specified after the ED command. You can type and save your SELECT statement in the .sql file.

3. Save the SELECT statement to a script file.

4. Edit the script file to add the iSQL*Plus commands by typing the following at the SQL prompt:

   SQL> ED script_file

   This opens the script file that contains the SELECT statement. Add formatting commands, to your requirement, before the SELECT statement.

5. Ensure that a RUN character, which is either a semicolon (;), if specified on the same line, or a slash (/), if specified on the next line, follows the SELECT statement.

6. Add the format clearing iSQL*Plus commands after the RUN character. Alternatively, you can create a separate reset file that stores all the clear format commands.

7. Save the script file with the changes.

8. Load the script file and execute the script file by typing the following iSQL*Plus command at the SQL prompt:

   SQL>@script_file
You will see the output that implements all the format commands. In addition to the above steps, you need to follow certain guidelines, as listed below:

- You can have blank lines between iSQL*Plus commands in a script file.
- If you have a lengthy command, you can continue the command on the next line by ending the first line with a hyphen (-).
- You can abbreviate iSQL*Plus commands unlike SQL statements.
- Always restore the original iSQL*Plus setting by resetting commands at the end of the script file.

The following example shows how to create a script file to run a report that provides employee data using iSQL*Plus commands. Create a script file to create a report that displays department-wise, title-wise employee data. Add a centered, two-line header “Employee Details” and a centered footer “End Of Report”. Rename the TITLE column to “Job Title”, FNAME to “First Name”, and LNAME to “Last Name”, and split headings over two lines. Display the salary in the $9,999.00 format and display “No commission” and “No Manager” for any null value in the COMMISSION and MGR_ID columns, respectively.

Create a script file, FastCERT_demo01.sql, by typing the following command at the SQL prompt.

```sql
SQL> ED FastCERT_demo01
```

You have created a script file where you can store SELECT statements along with iSQL*Plus commands. Type the following commands in the script file and execute the script file:

```sql
SET ECHO OFF
SET FEEDBACK OFF
SET VERIFY OFF
SET PAGESIZE 30
SET LINESIZE 100
BREAK ON DEPT_ID ON TITLE
COLUMN SALARY FORMAT $9,999.00
COLUMN COMMISSION FORMAT 99999999999999 JUSTIFY LEFT NULL "No Commission"
COLUMN FNAME HEADING 'FIRST |NAME'
COLUMN LNAME HEADING 'LAST |NAME'
COLUMN TITLE HEADING 'JOB |TITLE'
COLUMN MGR_ID FORMAT 9999999999 JUSTIFY LEFT NULL "No Manager"
TTITLE "Employee|Details"
BTITLE "End  Of | Report"
SELECT dept_id, title, emp_id, fname, lname, salary, mgr_id, commission
FROM employee
ORDER BY dept_id, title;
CLEAR columns
CLEAR BREAKS
TTITLE OFF
BTITLE OFF
SET ECHO ON
```
Save the above script file and execute it by typing the following command:

```sql
SQL> @KapCERT_demo01
```

The output is:

Mon Jun 02 page 1

**Employee Details**

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>JOB TITLE</th>
<th>EMP_ID</th>
<th>FIRST NAME</th>
<th>LAST NAME</th>
<th>SALARY</th>
<th>MGR_ID</th>
<th>COMMISSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>CLERK</td>
<td>3534</td>
<td>ERICH</td>
<td>JOHNSON</td>
<td>$1,300.00</td>
<td>3382</td>
<td>No Commission</td>
</tr>
<tr>
<td></td>
<td>MANAGER</td>
<td>3382</td>
<td>ROGER</td>
<td>ALMEIDA</td>
<td>$2,450.00</td>
<td>3439</td>
<td>No Commission</td>
</tr>
<tr>
<td></td>
<td>PRESIDENT</td>
<td>3439</td>
<td>JOHN</td>
<td>HOUSTON</td>
<td>$5,000.00</td>
<td></td>
<td>No Commission</td>
</tr>
</tbody>
</table>

End Of Report

The BREAK command creates a break in the Dept_id and Title columns. TTITLE sets the heading of the report to “Employee Details” and BTITLE sets the footer as “End Of Report”. The vertical bar inserts a line feed within the title. The headings for the Fname, Lname, and Title columns have been changed using the COLUMN command. The Commission and Mgr_id columns display “No Commission” and “No Manager,” respectively, in place of null values. The SET commands sets the iSQL*Plus environment.
Review Checklist: Producing Readable Output with iSQL*Plus

- Use the single-ampersand substitution variable to accept column values and table and column names from users.
- Identify the purpose of the double-ampersand substitution variable.
- Control the display of a column using the iSQL*Plus COLUMN command.
- Define the various options available with the FORMAT option of the COLUMN command.
- Create a script file to run an iSQL*Plus report.
Manipulating Data
Describe Each DML Statement

Scope

Manipulate data in the database tables by adding new rows, modifying the existing rows, and removing the existing rows from a table using the INSERT, UPDATE, and DELETE commands, respectively. Know the purpose of a MERGE statement. Know the definition of Data Manipulation Language (DML) and identify the DML statements.

Focused Explanation

So far, you have accessed database tables as an end user. You have been retrieving data using the Data Query Language statements. You can manipulate data present in database tables according to your requirements. You can insert new rows in a table, change the existing data in a table, or remove data from the table by using the INSERT, UPDATE, and DELETE statements, respectively.

INSERT

The INSERT statement allows you to INSERT new row(s) in a table. In the following example, you use the INSERT statement to insert a row in the DEPARTMENT_COPY table:

```
INSERT INTO department_copy (dept_id, name, location)
VALUES (1005,'KNOWLEDGE_DEP','NEW YORK');
```

The above statement creates a row in the DEPARTMENT table with the values 1005 for the DEPT_ID column, ‘KNOWLEDGE_DEP’ for the NAME column, and ‘NEW YORK’ for the LOCATION column. In the INTO clause of the INSERT statement, you specify the table name followed by a list of columns specified within parentheses. This list specifies the names of columns in which you need to insert values. The VALUES clause consists of the list of values that need to be inserted in the columns listed in the INTO clause.

UPDATE

You can modify the existing rows using the UPDATE statement. The following example updates the salary of employees working in department 1002 to value 2500 in the EMPLOYEE table.

```
UPDATE employee
SET salary = 2500
WHERE dept_id = 1002;
```

You can omit the WHERE clause. In case you omit the WHERE condition, the statement updates all rows of the employee table.
DELETE

Similarly, you can use the DELETE statement to remove data from database tables. The following example deletes rows, which belong to employees of department 1002, in the EMPLOYEE table:

```sql
DELETE FROM employee
WHERE dept_id=1002;
```

This deletes all rows that belong to department number 1002. In case you omit the WHERE condition, the above statements delete all rows from the EMPLOYEE table. Using the statements, you can perform insert, update, or delete operations in a table simultaneously.

MERGE

The MERGE statement allows you to conditionally insert some rows or update a few rows in a single operation. The decision is taken based on the join condition; rows that are already present in the target table matching the join condition are updated; if the rows are not present, the rows are inserted in the target table. This feature allows you to provide multiple UPDATE statements. The following example shows the syntax to use MERGE...INTO to insert values from the DEPARTMENT table in the DEPARTMENT_COPY table if matching records are not found in the target table or to update values in DEPARTMENT_COPY if corresponding records are found.

```sql
MERGE INTO department_copy c USING department d
ON (c.dept_id = d.dept_id)
WHEN MATCHED THEN
UPDATE SET c.name = d.name, c.location = d.location
WHEN NOT MATCHED THEN
INSERT VALUES(d.dept_id, d.name, d.location);
```

You can use MERGE statements in data warehousing where you commonly insert or update large amounts of data.

You can manipulate data in the database using DML. Adding new rows to a table, modifying the existing rows in a table, or removing existing rows from a table executes a DML statement. DML statements are statements that allow you to manipulate data in tables. In other words, INSERT, UPDATE, DELETE, and MERGE are DML statements. You cannot use DML to modify the structure of database tables.
Insert Rows into a Table

Scope

Identify the syntax of the INSERT clause. Understand the requirements of the VALUES clause in an INSERT statement. Identify SQL statements that insert data into a table and meet a list of requirements.

Focused Explanation

DML statements manipulate data in database tables. An example of a DML statement is an INSERT statement. You can add new rows to a table by using the INSERT statement. The general syntax of the INSERT statement is:

```
INSERT INTO table [(col1, col2)]
VALUES (val1, val2);
```

In the syntax, table is the name of the table, col1, col2... are the names of columns in the table to populate, and val1, val2... are the corresponding values of the columns. The INSERT with the VALUES clause inserts only one row at a time into a table. The following example shows how to insert values into the DEPARTMENT table using the INSERT statement:

```
INSERT INTO department (dept_id, name, location)
VALUES (1005,’KNOWLEDGE_DEP’,’NEW YORK’);
```

You can insert a new row that contains values for each column. In this case, the column list is not required in the INSERT clause. However, if you are omitting the column list in the INSERT clause, ensure that you list the values according to the default order of columns in the table and provide a value for each column. In the following example, the column list is omitted and values are inserted keeping the default order of columns in the table in mind:

```
INSERT INTO department
VALUES (1005,’KNLGD_DEP’,’NEW YORK’);
```

You must include the character and date values within parentheses. The following example inserts the employee id, the first name, the join date, the department id and the salary of the employee:

```
INSERT INTO employee (emp_id, fname, joindate, dept_id, salary)
VALUES (3590,’DEN’,’01-SEP-83’,1003,2800);
```
You must list the columns in the INSERT clause if you are not inserting values in all columns or in the default order of columns in the table. When creating a table, if a column is provided a DEFAULT value, you can specify that the column use the DEFAULT value in the VALUES clause. For example:

```
INSERT INTO employee (emp_id, fname, lname, salary, title, commission, mgr_id, dept_id, joindate) 
VALUES (3449, 'MICHAEL', ' ', 2600, DEFAULT, NULL, NULL, 1001, '09-DEC-81');
```

In the above example, the DEFAULT keyword specifies the column title to insert the DEFAULT value that has been provided when creating the table. If no default value has been provided to the column, the column will have null as the default value. You can explicitly provide null values by either specifying the NULL keyword or providing a null character or blank space within single quotes ("'"), as shown in the above example. The above example inserts a null value for the LNAME, COMMISSION, and MGR_ID columns.

Using Substitution Variables in the INSERT Statement

You can prompt the user for values that need to be inserted in columns by using substitution variables. You can use the single-ampersand and the double-ampersand substitution variables to accept values from the user and insert the values in the table. For example:

```
INSERT INTO employee (emp_id, fname, salary, dept_id, title) 
VALUES (&emp_id, '&fnm', &sal, &dep_id, '&title');
```

The values that you input for these substitution variables are then substituted into the statement. You can save the above statement in a script file and run the script file repeatedly but supply a different set of values each time.

Using a Subquery in the INSERT Statement to Copy Rows from Another Table

You can copy values from one table to another using a subquery, instead of the VALUES clause in the INSERT statement, and insert the values in another table. The subquery retrieves value from one table that needs to be inserted in another table. The following statement inserts values from the EMPLOYEE table into the EMPLOYEE_COPY table using a subquery in an INSERT statement:

```
INSERT INTO employee_copy (emp_id, fname, title, dept_id, salary) 
SELECT emp_id, fname, title, dept_id, salary 
FROM employee 
WHERE dept_id = 1003;
```

The number of columns and data types in the column list of the INSERT clause must be the same as the number of values and their data types in the subquery. In place of the VALUES clause, you may use the subquery.
You can also use the subquery in the INTO clause. You can replace the table name and the column list in the INTO clause with a subquery, as follows:

```sql
INSERT INTO (SELECT emp_id, fname, salary, dept_id, title
FROM employee_copy)
VALUES (3211, 'BEN', 3000, 1002, 'MANAGER');
```

You can put a check on the values to be inserted while using the subquery in the INTO clause by using the WITH CHECK OPTION. For example:

```sql
INSERT INTO (SELECT emp_id, fname, salary, dept_id
FROM employee_copy
WHERE salary > 1500 WITH CHECK OPTION)
VALUES (3201, 'ALLEN', 950, 1002);
```

However, the above statement is illegal because the value inserted for the SALARY column is 950, which is less than the value, 1500, specified in the subquery. The WITH CHECK OPTION clause does not allow any value, which would produce rows that are not included in the subquery, to be inserted. As a result, you will encounter the following error when you execute the above INSERT statement:

```
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```
Update Rows in a Table

Scope

Modify existing data using UPDATE statement. Identify the proper syntax for UPDATE statements. Update a value based on a value in another table. Update multiple columns in a table.

Focused Explanation

You can modify or change the existing data using the UPDATE statement. The general syntax to update rows in a table is:

```
UPDATE table SET col1 = value [, col2 = value,] [WHERE condition];
```

In the above syntax, table is the name of the table, col1 is the name of the column to be updated, value is the corresponding value or a subquery for the column, and condition identifies the rows that need to be updated. The following example modifies the salaries of employees in department 1003:

```
UPDATE employee
SET salary = salary + salary * 0.12
WHERE dept_id = 1003;
```

You can omit the WHERE condition. In this case, the statement would modify all rows of the employee table. You can include subqueries in the SET clause. You can set columns to a value retrieved from a subquery. For example:

```
UPDATE employee
SET dept_id = (SELECT dept_id
               FROM employee
               WHERE emp_id = 3254)
WHERE salary BETWEEN 1500 and 2500;
```

The above example updates the DEPT_ID column of employees who earn salaries between 1,500 and 2,500 to a value equal to that of the department number of employee 3254. The subquery returns DEPT_ID of employee 3254. You can have more than one subquery in an INSERT statement. The following example uses two subqueries, one in the SET clause and another in the WHERE clause:

```
UPDATE employee
SET mgr_id = (SELECT mgr_id
              FROM employee
              WHERE emp_id = 3388)
WHERE salary >ANY (SELECT salary
                    FROM employee
                    WHERE dept_id=1003);
```
While using the subqueries, you must ensure that you use the appropriate comparison operator, depending on the number of rows returned by the subquery. For example, the following subquery returns an error message as the subquery returns multiple rows and the comparison operator used is a single-row comparison operator.

```
UPDATE employee
SET salary = (SELECT salary
    FROM employee
    WHERE dept_id = 1001);

SET salary = (SELECT salary FROM employee WHERE dept_id = 1001);

ERROR at line 2:
ORA-01427: single-row subquery returns more than one row
```

You can specify the column to be updated to its default value specified when creating the table. If the default value has not been specified for a column during its creation, a null value is inserted. The following example sets the MGR_ID column to its default value:

```
UPDATE employee
SET mgr_id = DEFAULT
WHERE dept_id=1002;
```

Updating rows based on another table

You can update rows in a table based on values from another table using subqueries. The following example updates the employee table based on values in the department table:

```
UPDATE employee SET dept_id = (SELECT dept_id
    FROM department
    WHERE name = 'RES_DEV')
WHERE salary > 1500;
```

Updating Multiple Columns

You can update multiple columns in a single UPDATE statement. You require only one SET clause to update multiple columns. You must separate columns that need to be updated by a comma.

```
UPDATE employee_copy
SET salary = salary + salary * .12, title = (SELECT title
FROM employee
WHERE emp_id = 3121),
    dept_id = (SELECT dept_id
    FROM department
    WHERE name = 'ACCOUNTS');
```
Delete Rows from a Table

Scope

Remove existing rows from a table using DELETE statements. Identify the proper syntax of DELETE statements. Identify DELETE statements that violate child rules. Use a subquery in the WHERE clause of a DELETE statement. Know the effects of DELETE statements.

Focused Explanation

You can remove existing rows from a table using DELETE statements. The general syntax for the DELETE statement is:

```
DELETE [FROM] table
[WHERE condition];
```

In the above syntax, table is the name of the table and condition is the WHERE condition that specifies the rows to be deleted. The following example deletes the records of all clerks:

```
DELETE FROM employee
WHERE title = 'CLERK';
```

All rows of a table are deleted if you omit the WHERE condition. A DELETE statement without a WHERE condition deletes table data and not the table structure. As a result, the following statement does not delete the structure of the EMPLOYEE table:

```
DELETE employee;
```

You cannot use column names in DELETE statements because the DELETE operation is performed on rows. As a result, an entire row is deleted and not a column value. You cannot perform deletion on tables that are linked to another table. In other words, if you try to delete a record with a value tied to an integrity constraint, an error is returned. For example, the department number of the DEPARTMENT table is used as a FOREIGN KEY in the EMPLOYEE table. Hence, if you attempt to delete the parent record that has the child record, you obtain the following error message:

```
DELETE FROM department
WHERE dept_id = 1001;
*  
ERROR at line 1:
ORA-02292: integrity constraint (SCOTT.EMP_FOREIGN_KEY) violated - child record found
```
You can use subqueries in the WHERE clause of DELETE statements. For example,

```
DELETE FROM employee
WHERE dept_id = (SELECT dept_id
                   FROM department
                   WHERE name = 'ACCOUNTS');
```

You must ensure that you use appropriate comparison operators when you use subqueries.
Merge Rows in a Table

Scope

Identify valid uses of a MERGE statement. Merge data from one table with data from another table into one target table using the MERGE statement. Identify the syntax of MERGE statements.

Focused Explanation

The MERGE statement allows you to conditionally insert or update data in a database table. The decision is taken based on the join condition provided in the ON clause; rows already present in the target table, which match the join condition, are updated; if the rows are not present, the rows are inserted in the target table. This feature allows you to avoid multiple UPDATE statements. The general syntax for the MERGE statement is:

```
MERGE INTO tab_name AS tab_alias
USING (table/view/subquery) AS alias
ON (join_condition)
WHEN MATCHED THEN
UPDATE SET col1 = val1, col2 = val2
WHEN NOT MATCHED THEN
INSERT (col_list)
VALUES (col_values);
```

In the above syntax, the INTO clause specifies the target table, the USING clause specifies the source table, the ON clause specifies the join condition based on which the merge option either updates or inserts values, and the WHEN MATCHED and WHEN NOT MATCHED clauses provide the response of the server to the join condition. The following example shows the syntax to use MERGE...INTO to insert values from the EMPLOYEE table into the EMPLOYEE_COPY table if matching records are not found in the target table or to update values in EMPLOYEE_COPY if corresponding records are found.

```
MERGE INTO employee_copy c
USING employee e ON(c.emp_id = e.emp_id)
WHEN MATCHED THEN
UPDATE SET c.fname=e.fname, c.lname=e.lname, c.dept_id = e.dept_id, c.title = e.title
WHEN NOT MATCHED THEN
INSERT VALUES e.emp_id, e.fname, e.lname, e.dept_id, e.title);
```

The above example matches EMP_ID in the EMPLOYEE_COPY table to EMP_ID in the EMPLOYEE table. If a match is found, the row in the EMPLOYEE_COPY table is updated to match the row in the EMPLOYEE table. If the row is not found, the row is inserted into the EMPLOYEE_COPY table.

You can use MERGE statements in data warehousing where large amounts of data are commonly inserted or updated.
Control Transactions

Scope

Define a database transaction. Identify statements that cause an implicit commit. Identify the characteristics of a transaction. Identify the effects of the ROLLBACK and SAVEPOINT statements on a transaction.

Focused Explanation

A database transaction consists of DML statements that constitute one consistent change to the data, one Data Control Language (DCL) statement, or one Data Definition Language (DDL) statement. The transactions give you more control when changing data, ensuring data consistency in the event of a user process or system failure. For example, if you transfer some money from one account to another, the process should include debit to one account and credit to another account. Both actions should either succeed or fail. A database transaction begins when the first DML statement is executed and ends with any one of the following:

A COMMIT or a ROLLBACK statement is issued.

A DCL statement, such as GRANT or REVOKE, is issued.

A DDL statement, such as CREATE, DROP, TRUNCATE, or ALTER is issued.

The user exits the session.

The system crashes.

A DDL or a DCL statement issues an implicit commit and implicitly ends a transaction. DML statements do not issue an implicit commit or rollback. You have to explicitly issue a COMMIT or a ROLLBACK statement to commit changes or revert changes, respectively. A COMMIT statement is issued implicitly when a user exits a session normally without having a system crash. You can explicitly issue a COMMIT or a ROLLBACK, as follows:

```
SQL> COMMIT;
SQL> ROLLBACK;
```

State of Data Before a COMMIT or a ROLLBACK

The current user can review the result of the DML statement by querying the table, while other users cannot view the results of the DML statement made by the current user before a COMMIT or ROLLBACK statement. The Oracle server ensures read consistency because each user sees the data as it existed at the last commit. The rows are locked for DML statements for other users.
State of Data after a COMMIT

Once you issue a COMMIT, the changes are made permanent in the database. All users can view the changes and the state of data before the commit is permanently lost. All locks are released and the rows are available to other users for manipulation.

State of Data after a ROLLBACK

You can undo non-committed DML statements by issuing a ROLLBACK. All pending changes are discarded. The previous state of data is restored and locks are released.

You cannot rollback changes after you have committed them. Every change in data is temporary until you commit that transaction.

Statement-Level Rollback

If one of the statements in a transaction fails during execution, only that statement is rolled back. After every statement in a transaction, the Oracle server implicitly creates a savepoint. As a result, if the statement fails, only that statement is rolled back and the changes made by previous DMLs are not discarded. The Oracle server issues an implicit commit before and after every DDL and DCL statement. As a result, even if the DDL or DCL statement fails, you cannot rollback the previous statement because that has already been committed.

Read Consistency

The Oracle server ensures read consistency to provide a consistent view of data at all times by issuing locks on the data. There are two ways to access the database, READ operations and WRITE operations. The READ operation constitutes a SELECT statement and the WRITE operations constitute INSERT, UPDATE, and DELETE statements. Database readers and writers are ensured a consistent view of data because the readers do not view the data that is in the process of being changed. Rows are locked for other users when a user is trying to make changes in the rows. The rows remain locked for others until the current user issues a COMMIT or ROLLBACK.

Savepoint

A savepoint is like a marker or checkpoint that divides the transaction into smaller sections so that you can rollback up to a particular section. Oftentimes, a single transaction is executing many DMLs. At the end of the transaction, when you issue a rollback, all DML statements belonging to that transaction are discarded. You can rollback a set of statements by issuing a savepoint after each DML statement.
For example, you have a transaction that consists of a few INSERTs, DELETEs, and UPDATEs, as follows:

```
SQL> INSERT...
SQL> UPDATE...
SQL> INSERT...
SQL> DELETE...
SQL> DELETE...
```

Now, if you issue a rollback, all the above DML statements would be discarded. You can roll back up to a certain point if you create a marker or savepoint. You can create a savepoint and rollback up to that savepoint. For example:

```
SQL> INSERT...
SQL> Savepoint A;
Savepoint created.
SQL> UPDATE...
SQL> INSERT...
SQL> Savepoint B;
Savepoint created.
SQL> DELETE...
SQL> Savepoint C;
Savepoint created.
SQL>DELETE...
```

Now, if you need to roll back only the last DELETE, you can specify the following command:

```
SQL> Rollback to C;
Rollback complete.
```

You cannot rollback in between transactions. In other words, you cannot rollback transactions between savepoints A and B. If you issue ROLLBACK TO A, all statements until savepoint A will be rolled back.
Review Checklist: Manipulating Data

- Define a database transaction.
- Identify statements that issue an implicit COMMIT.
- Know the state of data before and after a COMMIT.
- Know the state of data before and after a ROLLBACK.
- Define statement-level rollback.
- Define read consistency and know how the Oracle server ensures read consistency to users.
- Identify the significance of savepoints.
Creating and Managing Tables
Describe the Main Database Objects

Scope

Identify database objects. Define a table, view, sequence, index, and synonym.

Focused Explanation

A database can contain several data structures. Each structure should be outlined in the design of the database so that it can be created during the development stage of the database. A few database objects are briefly described in this section:

Table

A table is a basic unit of storage, which is composed of rows and columns. Tables can be created at any time, even when users are using the database. Any user with a CREATE TABLE privilege can create a table.

View

A view logically represents subsets of data from one or more tables. You can restrict a user’s access to data in a table by using views. A view does not have a structure of its own and is stored as a SELECT statement in the database.

Sequence

A sequence is a user-defined object to generate unique numeric values. You can use a sequence to create a PRIMARY KEY value, which must be unique for each row. Sequences are stored and generated independently of tables.

Index

An index is a database object that the Oracle server uses to speed up retrieval of rows by using a pointer. You can create indexes explicitly and automatically. An index provides faster access to rows in a table. The Oracle server manages the index automatically.

Synonym

A synonym is a database object that provides a better name to refer to objects. You can use synonyms to shorten lengthy names or to ease the reference to an object owned by another user. You can create synonyms for a table, view, sequence, procedure and other objects.
Create Tables

Scope

Identify table and column naming conventions and correct table and column names. Identify the correct CREATE TABLE syntax and identify errors in CREATE TABLE statements. Understand the need for schema qualifiers. Define number and character data types. Define date columns that use SYSDATE as the DEFAULT value. Define a DEFAULT value for a VARCHAR2 column. Know the table creation rule and characteristics.

Focused Explanation

A table is a database object created by users to store data in the form of rows and columns. A table is the basic unit of storage. You can create a table at any time, even while users are accessing the database. A database table consists of column that needs to be defined when creating the table. You usually insert rows after creating the table using the INSERT clause. You can have up to 1,000 columns in a table. A table must conform to some standard database object-naming conventions, which are:

- Table names and column names must begin with a letter and be a maximum of 30 characters long.
- Names must only contain the A-Z, a-z, 0-9, underscore (_), $, and # characters.
- You cannot have two objects with the same name owned by the same user.
- Names must not be an Oracle server reserved word, such as DEFAULT, SYSDATE, TABLE, NUMBER, PACKAGE, and so on.

The CREATE TABLE Syntax

You can create a table to store data if you have the CREATE TABLE privilege and a storage area allocated to create objects. You can create a table using the CREATE TABLE statement, which is a DDL statement. The general syntax to create a table is:

```
CREATE TABLE [schema.] table (column data type [DEFAULT expr] [, ...]);
```

In the above syntax, schema is the same as the owner's name, table is the name of the table, DEFAULT expr specifies the default value if the value is not provided in the INSERT list, column is the name of the column, and data type is the data type and length of the column.
The following example creates a table, named Location:

```sql
CREATE TABLE location
(id NUMBER(4),
 name VARCHAR2(20));
```

The above example creates a table, called Location, with two columns, Id, which is a numeric column; and Name, which is a character column. The table is created in the owner’s schema. A schema is a collection of objects, which has the same name as the user. Schema objects include tables, views, sequences, synonyms, procedures, indexes, and so on. You can access a table that belongs to another user by prefixing the owner’s name to the table. For example, you are USER_1 and you need to access USER_2's EMPLOYEE table; specify the following to retrieve data from the EMPLOYEE table:

```sql
SELECT *
FROM user_2.employee;
```

In case you try to create a table with a name that is the same as the name of an existing table, you obtain the following error:

```sql
CREATE TABLE EMPLOYEE
*  
ERROR at line 1:
ORA-00955: name is already used by an existing object
```

You have to drop the existing table and then create the table that has the same name with a new definition. Oracle does not provide a command, such as CREATE or REPLACE TABLE.

**The DEFAULT Option**

You can provide a default value for a column by using the DEFAULT option. A default value is the value inserted in a column when the user does not provide a value for that column in the INSERT statement. The default value can be a literal value, an expression, or an SQL function, such as SYSDATE, USER, and so on. You cannot provide another column’s name as the default value. The DEFAULT expression must match the data type of the column. You can specify the DEFAULT value for a column while creating the table, as follows:

```sql
CREATE TABLE orders
(id NUMBER (4),
 customer_Id NUMBER (4),
 order_date DATE DEFAULT SYSDATE,
 ship_date DATE,
 total NUMBER (7,2),
 pay_type VARCHAR2 (20) DEFAULT 'CASH');
```
In the previous example, you have created an ORDERS table that has six columns. The columns are:

- **ID**, which is a Number column with a precision of four digits, as mentioned within parentheses;
- **CUSTOMER_ID**, which is also a Number column with a precision of four digits;
- **ORDER_DATE**, which is a Date column with SYSDATE as the default value;
- **SHIP_DATE**, which is a Date column;
- **TOTAL**, which is a Number column with a precision of seven digits, which is the total number of digits and a scale of 2, which is the number of digits to the right of the decimal point;
- **PAY_TYPE**, which is a Character column of the VARCHAR2 data type, is 20 characters long, with ‘CASH’ as the DEFAULT value.

A character column can be of CHAR or VARCHAR2. CHAR is fixed-length character data, with a minimum, default size of 1 and a maximum size of 2000. VARCHAR2 is a variable-length character data, with a minimum size of 1 and a maximum size of 4000.

You can create a table at any time. You cannot rollback the creation of a table because CREATE TABLE issues an implicit commit. You have to drop the table to revert the CREATE TABLE statement. The tables are, by default, created in the owner’s schema unless you specify the name of another schema in the CREATE TABLE statement. The tables are not accessible to all users, by default. You have to explicitly grant privileges to other users to access your table using the DCL GRANT command. You can confirm the creation of the table by describing the structure of the table you created using the iSQL*Plus DESCRIBE command, as follows:

```
DESCRIBE orders
```

The output will be:

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>NUMBER (4)</td>
<td></td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
<td>NUMBER (4)</td>
<td></td>
</tr>
<tr>
<td>ORDER_DATE</td>
<td>DATE</td>
<td></td>
</tr>
<tr>
<td>SHIP_DATE</td>
<td>DATE</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>NUMBER (7,2)</td>
<td></td>
</tr>
<tr>
<td>PAY_TYPE</td>
<td>VARCHAR2 (20)</td>
<td></td>
</tr>
</tbody>
</table>
Another way to confirm table creation is by querying the data dictionary. A data dictionary stores data about the data in the database. It is a collection of information about database objects. The Oracle server creates and maintains the data dictionary. The user SYS owns all data dictionaries. Many data dictionaries that store information about different objects are available. You can confirm table creation by querying the data dictionary view USER_TABLES in the following manner:

```sql
SELECT table_name
FROM user_tables
WHERE table_name = 'ORDERS';
```

The above statement confirms the creation of your table.
Describe the Data Types that Can be Used when Specifying Column Definitions

Scope

Know the different data types for columns. Know the different DATETIME data types. Know the default fractional seconds precision of the TIMESTAMP data type. Define the ROWID data type.

Focused Explanation

In the CREATE TABLE statement, you list the columns of the table along with the table’s data type and length. SQL provides the following data types:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR2 (len)</td>
<td>Is variable-length character data (a max len must be specified). The minimum value for len is 1 byte and the maximum value for len is 4,000 bytes.</td>
</tr>
<tr>
<td>CHAR (len)</td>
<td>Is fixed-length character data of length len bytes. The minimum, default value is 1 and the maximum value is 2,000 bytes.</td>
</tr>
<tr>
<td>NUMBER (pr, sc)</td>
<td>Is a number with precision pr and scale sc. The range of precision is from 1 to 38 and scale is from –84 to 127.</td>
</tr>
<tr>
<td>DATE</td>
<td>Are date and time values</td>
</tr>
<tr>
<td>LONG</td>
<td>Is variable length character data of length up to 2 giga bytes (GB)</td>
</tr>
<tr>
<td>CLOB</td>
<td>Is character data up to 4 GB</td>
</tr>
<tr>
<td>RAW and LONG RAW</td>
<td>Is raw binary data</td>
</tr>
<tr>
<td>BLOB</td>
<td>Is binary data up to 4 GB</td>
</tr>
<tr>
<td>BFILE</td>
<td>Is binary data stored in an external file</td>
</tr>
</tbody>
</table>

There are some restrictions on a few of these data types. A LONG column is not copied when a table is created using a subquery. You cannot have more that one long or long raw column per table. You cannot include a LONG column in a GROUP BY or an ORDER BY clause.
Datetime Data Types

Apart from the above mentioned data types, there are other datetime data types. Oracle9i has introduced new datetime data types. Some enhancements have been made to time zones and the local time zone in Oracle9i. The following are the datetime data types available in Oracle9i:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP</td>
<td>Is a date with fractional seconds</td>
</tr>
<tr>
<td>INTERVAL YEAR TO MONTH</td>
<td>Stored as an interval of years and months</td>
</tr>
<tr>
<td>INTERVAL DAY TO SECONDS</td>
<td>Stored as an interval of days, hours, minutes, and seconds</td>
</tr>
</tbody>
</table>

**TIMESTAMP**

The TIMESTAMP data type is an extension to the DATE data type. TIMESTAMP stores the year, month, day of the date, hours, minutes, and seconds as well as the fractional second values. The syntax for the TIMESTAMP data type is:

```
TIMESTAMP [(fractional_seconds_precision)]
```

In the syntax, fractional_seconds_precision specifies the number of digits in the fractional part of the seconds datetime column and can have an integer value in the 0 to 9 range. This parameter is optional and has a default value of 6. For example;

```
CREATE TABLE emp
(id NUMBER,
 name VARCHAR2(20),
...
 hiredate TIMESTAMP(7));
```

The value in the HIREDATE column would be displayed in the following format:

```
16-JUN-81 12:00:00.0000000 P.M
```

**TIMESTAMP WITH TIME ZONE**

TIMESTAMP WITH TIME ZONE, which is an extension of the TIMESTAMP data type includes the time zone displacement. The time-zone displacement is the difference in hours and minutes between the local time and Coordinated Universal Time (UTC), formerly Greenwich Mean Time. The syntax for TIMESTAMP WITH TIME ZONE is:

```
TIMESTAMP [(precision)] WITH TIME ZONE
```
The format of a value that belongs to a column of the TIMESTAMP WITH TIME ZONE data type is:

'1981-06-16 12:00:00 -6:00'

**TIMESTAMP WITH LOCAL TIME ZONE**

TIMESTAMP WITH LOCAL TIME ZONE is the same as TIMESTAMP WITH TIME ZONE, except that when you insert a value in the database column, it is normalized to the database time zone and does not store the displacement in the column. When you retrieve the value, Oracle returns it in your local session time zone.

**INTERVAL YEAR TO MONTH**

The INTERVAL YEAR TO MONTH data type stores the intervals of years and months. The syntax is:

```
INTERVAL YEAR [(precision_of_year)] TO MONTH
```

In the above syntax, precision_of_year defines the digits in the year field and can take integer values in the 0 to 4 range. The default value is 2.

For example,

```
INTERVAL '115-2' YEAR (1) TO MONTH
```

The above output indicates 115 years and 2 months.

```
INTERVAL '200' MONTH (2)
```

The above indicates an interval of 2 months.

```
INTERVAL '311' YEAR (3)
```

The above indicates 311 years and 0 months.

**INTERVAL DAY TO SECOND**

The INTERVAL DAY TO SECOND data type stores the intervals of days, hours, minutes, and seconds. The syntax is:

```
INTERVAL DAY [(lead_pr)] TO SECOND [(fractional_seconds_precision)]
```

In the above syntax, lead_pr and fractional_seconds_precision define the digits in the days and seconds fields, respectively. The two parameters lead_pr and fractional_seconds_precision can take integer values in the 0 to 9 range. The default value is 2 for lead_pr and 6 for fractional_seconds_precision.
ROWID

Each row in the database table has an address. You can retrieve a row’s address by querying the pseudocolumn, ROWID. A pseudocolumn is similar to a column that is not stored in a table. You can select from a pseudocolumn, but you cannot perform DML operations on it. The values of the pseudocolumn ROWID are hexadecimal strings that represent the address of the row. The strings have the data type ROWID.
Alter Table Definitions

Scope

Know the uses of the ALTER TABLE statement. Know the effects of using the ALTER TABLE statement to modify a column's DEFAULT value. Identify the proper syntax for the ALTER TABLE statement when modifying a column's precision. Know when it is possible to modify a column's size. Know the proper syntax for the ALTER TABLE SET UNUSED statement.

Focused Explanation

After you have created a table, you may need to modify the table structure by adding a column, removing a column, or modifying the definition of the existing column. You can do this using the ALTER TABLE statement. You can use the ALTER TABLE statement to:

- Add a new column.
- Modify an existing column.
- Define a default value for the new column.
- Set a column as unused.
- Drop a column.

Adding a New Column

The syntax for the ALTER TABLE statement to add a new column is:

```
ALTER TABLE table
ADD (col data type [DEFAULT expr] [, col data type]...);
```

You can use the ADD clause to add a new column. For example;

```
ALTER TABLE department_copy
ADD (employee NUMBER (7));
```

The above statement adds the EMPLOYEE column to the DEPARTMENT_COPY table's definition. You cannot specify the position of the column. The column is added to the end as the last column. If the table already contains data, the new column is initially assigned a null value for all rows.

Modifying an Existing Column

You can change the data type, size, and default value of a column using the ALTER TABLE statement. The syntax for the ALTER TABLE statement to modify an existing column is:

```
ALTER TABLE table
MODIFY (col data type [DEFAULT expr] [, col data type]...);
```
You can change or assign the default value of an existing column of the table, as shown in the following example:

```
ALTER TABLE employee
MODIFY dept_id NUMBER DEFAULT 1003;
```

The above example sets a default value, 1003, for the DEPT_ID column of the EMPLOYEE table. Similarly, you can set the DEFAULT value for the character and date columns.

You can change the data type of the column using the ALTER TABLE…MODIFY statement, as follows:

```
ALTER TABLE emp01
MODIFY job VARCHAR2(20);
```

The above statement changes the JOB column from the CHAR data type to the VARCHAR2 data type and is successful only if the JOB column contains null values.

You can increase the width or precision of a numeric value or the width of a character value. You can decrease the width of a column only if it contains null values or if the table does not contain any rows. The following example increases the width of the last name column of the EMPLOYEE table:

```
ALTER TABLE Employee
MODIFY lname VARCHAR2(35);
```

**The SET UNUSED Option**

You can set one or more columns as unused so that they can be dropped when system resources have lower demand. Setting a column unused removes the column from the table definition. Because this feature does not release the storage space of that column, it takes less response time to execute this command than the DROP clause. Once you set a column as unused, you cannot have access to that column. A SELECT query or a DESCRIBE command does not show that column anymore. The general syntax to set columns as unused is:

```
ALTER TABLE table
SET UNUSED (col);
```

OR

```
ALTER TABLE table
SET UNUSED COLUMN col;
```

You can set multiple columns as unused in a single statement using the SET UNUSED (col) clause. For example;

```
ALTER TABLE emp11
SET UNUSED COLUMN (mgr, comm);
```

The above example sets the MGR and COMM columns of the EMP11 table as unused. You can retrieve information about unused columns from the data dictionary view USER_UNUSED_COL_TABS.
Dropping a Column

The syntax for the ALTER TABLE statement to drop a column is:

```
ALTER TABLE table
DROP COLUMN col;
```

You can use the DROP clause to remove a column. For example;

```
ALTER TABLE emp11
DROP COLUMN job;
```

You can drop unused columns using the DROP clause, as follows:

```
ALTER TABLE emp11
DROP UNUSED COLUMN;
```
Drop, Rename, and Truncate Tables

Scope

Know the syntax and effects of dropping a table. Know how to rename a table using the RENAME statement. Know the effects of the TRUNCATE TABLE statement.

Focused Explanation

You can drop an entire table. The DROP TABLE statement removes data and the table definition from the database. When you drop a table, Oracle automatically performs the following operations:

- Deletes the data and structure in the table
- Commits all pending transactions
- Drops all indexes associated with the table

You can drop a table in the following manner:

    DROP TABLE emp11;

Once you drop a table, you cannot roll back the DROP TABLE statement because the DROP TABLE statement is implicitly committed.

Renaming the Table

You can change the name of a table, view, sequence, or synonym using the RENAME statement, as follows:

    RENAME dep11 TO dept;

You can execute the above statement only if you are the owner of the object you are renaming.

The TRUNCATE TABLE statement

Another DDL statement is the TRUNCATE TABLE statement, which removes all rows from the table and releases the storage space used by that table. You can truncate a table in the following manner:

    TRUNCATE TABLE dept;

The TRUNCATE TABLE statement removes data and releases storage space but does not remove the structure of the table, while the DELETE statement removes data, does not release storage space, and does not remove the structure. Being a DDL statement, you cannot roll back the TRUNCATE TABLE statement. You can execute the TRUNCATE TABLE statement only if you are the owner of the table you are truncating or you have been granted the DELETE TABLE privilege. The TRUNCATE TABLE statement is faster than the DELETE statement because no rollback information is generated for the TRUNCATE TABLE statement unlike for the DELETE statement.
Review Checklist: Creating and Managing Tables

- Define different database objects.
- Understand and implement table and column-naming conventions and rules for proper table and column names.
- Define numeric, character, and date data type columns.
- Use the DEFAULT keyword to specify a default value for the column.
- Know different data types for columns.
- Understand the syntax for the DATETIME data type.
- Alter the definition of the table using the ALTER TABLE statement.
- Add, modify, and remove a column using the ALTER TABLE statement.
- Set columns as unused using the SET UNUSED clause.
- Drop a table using the DROP TABLE statement.
- Rename a table using the RENAME TABLE statement.
- Truncate a table using the TRUNCATE TABLE statement and know the difference between TRUNCATE and DELETE.
Including Constraints
Describe Constraints

Scope
Define table constraints. Know valid Oracle constraints and their uses. Identify the characteristics of a constraint.

Focused Explanation
A table stores a related set of data. You can enforce rules on this data using constraints. Constraints are rules enforced on data to validate the data when a row is inserted, updated, or deleted. You can prevent the deletion of a table if there are dependents on other tables using the constraints. Oracle provides the following five types of constraints:

- NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY
- CHECK

Constraints enforce rules at the table level. You can use constraints to impose business rules on your table data.

NOT NULL
A NOT NULL constraint implies that a column defined as a NOT NULL column cannot contain null values. In other words, you can neither insert a null value in that column nor update the existing value to a null value for that column. For example, in the EMPLOYEE table, you can impose a rule that every employee must be assigned a department. As a result, you can define a NOT NULL constraint for the DEPT_ID column of the EMPLOYEE table so that the user is forced to specify a value for the DEPT_ID column of the EMPLOYEE table, assigning a department number for each employee.

UNIQUE
A UNIQUE constraint specifies that a column or combination of columns cannot have duplicate values for all rows in the table. You must insert unique values in that column. For example, you create a table, called PROJECT, which has a column, PROJ_NAME. You can impose a rule that two projects cannot have the same name by enforcing the UNIQUE constraint on the PROJ_NAME column. A unique column can contain null values because null values cannot be compared with anything.

PRIMARY KEY
A PRIMARY KEY uniquely identifies each row in a table. A PRIMARY KEY column cannot have duplicate values and must be provided with a definite value. In other words, a PRIMARY KEY is a combination of the UNIQUE and NOT NULL constraints. You can enforce the PRIMARY KEY constraint on one column or on a combination of columns. Each row value of the PRIMARY KEY column must be unique. For example, in the DEPARTMENT table, the DEPT_ID column is a unique identifier for each department. You cannot assign the same DEPT_ID value to two departments and each department must be assigned a value. As a result, you can define the DEPT_ID column as the PRIMARY KEY for the DEPARTMENT table.
FOREIGN KEY

A FOREIGN KEY establishes and enforces a referential relationship between a column or a combination of columns and a PRIMARY KEY column or a UNIQUE column of a referenced table, which can be the same or a different table. For example, in the EMPLOYEE table, you can assign a department number to an employee that exists in the DEPARTMENT table. As a result, the DEPT_ID column value of the EMPLOYEE table depends on the value of the DEPT_ID column of the DEPARTMENT table. You can implement this by creating the DEPT_ID column of the EMPLOYEE table as the FOREIGN KEY, which references the DEPT_ID column of the DEPARTMENT table. The DEPT_ID column is the PRIMARY KEY of the DEPARTMENT table. The FOREIGN KEY column must match an existing value in the parent table or be null, unless defined as NOT NULL.

CHECK

A CHECK constraint specifies a condition that the value must satisfy. Each row in the table must make the condition either TRUE or NULL. You can use a CHECK constraint mainly to impose a business rule, such as the salary of an employee cannot be less than 500. You can set the minimum salary for employees using the CHECK constraint, so that, if any user attempts to insert a salary value less than 500, the value is not accepted.

You can create these constraints while creating the table. You can also create constraints after creating the table using the ALTER TABLE statement. The constraint definition can be specified either at the table level or at the column level. A unique index is always created for a PRIMARY KEY and UNIQUE column. As a result, you can enforce rules on data using constraints and validate the data.
Create and Maintain Constraints

Scope

Identify the proper syntax to create constraints. View the details of the constraints in data dictionary views. Add a constraint to the existing table using the ALTER TABLE statement. Explicitly name a constraint. Disable or enable a constraint. Drop a constraint.

Focused Explanation

You can define constraints while creating the table. Constraints can be defined either at the table or column level. The general syntax to define a constraint while creating a table is:

```sql
CREATE TABLE [schema.] table
  (col1 data type [DEFAULT expr] [col1_constraint],
   ...
  [table_constraint] [,....]);
```

In the above syntax, schema is the owner’s name, table is the name of the table, col1 is the column name, DEFAULT expr specifies the DEFAULT value for the col1 column, data type is the column’s data type and length, col1_constraint is an integrity constraint defined at the column level, and table_constraint is an integrity constraint defined at the table level.

You can define constraints at two levels:

COLUMN level: References a single column and is defined within the specification for the owning column. The syntax is:
```
column [CONSTRAINT constraint_name ] constraint_type,
```

TABLE level: References one or more columns and is defined separately from the definitions of the columns in the table. The syntax is:
```
column,...
  [CONSTRAINT constraint_name] constraint_type (column_name,...) ,
```

In the syntaxes mentioned above, constraint_name is the name of the constraint and constraint_type is the type of the constraint.
Defining a NOT NULL Constraint

You can define the NOT NULL constraint only at the column level and not at the table level. The following example defines the title column of the Employee table as NOT NULL, which specifies that each employee must be assigned a job title:

```sql
CREATE TABLE employee
(emp_id NUMBER(4),
 fname VARCHAR2(20),
 ...,
 title VARCHAR2(20) NOT NULL,
 ...);
```

In the above example, the Oracle server provides a name for the NOT NULL constraint because you have not specified any name for the constraint.

Defining a UNIQUE Constraint

You can define the UNIQUE constraint either at the table or column level. In the following example, the UNIQUE constraint is defined for the PROJ_NAME column at the table level and a user-defined name is also specified for the constraint:

```sql
CREATE TABLE project
(proj_id_ID NUMBER(6),
 proj_name VARCHAR2(30),
 ..., CONSTRAINT name_uk UNIQUE (proj_name));
```

You can define a UNIQUE constraint on a combination of columns. This type of UNIQUE key is known as Composite Unique Key, which can be created only at the table level.

Defining a PRIMARY KEY Constraint

A PRIMARY KEY constraint uniquely identifies rows in a table. You cannot have more than one Primary Key column in a table but can have multiple UNIQUE columns. The PRIMARY KEY constraint can either be defined at the table or column level. You can have combinations of columns defined as a PRIMARY KEY, known as Composite PRIMARY KEY. For example, you can define the DEPT_ID column of the DEPARTMENT table as the PRIMARY KEY because each department must have a department ID and the value must be unique, as follows:

```sql
CREATE TABLE department
(dept_id NUMBER(4),
 name VARCHAR2(30) NOT NULL,
 ... CONSTRAINT dept_id_pk PRIMARY KEY (Dept_id));
```
Defining a FOREIGN KEY Constraint

A FOREIGN KEY constraint establishes a relationship between the FOREIGN KEY column of the child table and the PRIMARY or UNIQUE KEY column of the referenced or parent table. You can define a FOREIGN KEY constraint at either the column or table level. You must define the Composite FOREIGN KEY at the table level. The keywords associated with the definition of the FOREIGN KEY constraint are:

- **FOREIGN KEY**: Defines the column in the child table when defined at the table level. This keyword is not used if you define a FOREIGN KEY constraint at the column level.
- **REFERENCES**: Identifies the parent table and the parent table's PRIMARY or UNIQUE KEY column.
- **ON DELETE CASCADE**: Specifies that dependent rows in the child table must be deleted when the corresponding parent table row is deleted.
- **ON DELETE NULL**: Converts FOREIGN KEY values to null when the parent key value is deleted.

The following example creates a FOREIGN KEY constraint on the DEPT_ID column of the EMPLOYEE table at the table level, which references the DEPT_ID column of the DEPARTMENT table:

```sql
CREATE TABLE employee
(emp_id NUMBER(4),
...
department_id NUMBER(4),
CONSTRAINT dept_id_fk FOREIGN KEY (department_id) REFERENCES DEPARTMENT (department_id) ON DELETE CASCADE);
```

The above example defines the DEPT_ID column of the EMPLOYEE table as a FOREIGN KEY, which references the DEPT_ID column of the DEPARTMENT table for its value. ON DELETE CASCADE specifies that any deletion in the parent table would result in the deletion of corresponding rows in the child table. A table can contain multiple FOREIGN KEY columns.

Defining a CHECK Constraint

A CHECK constraint defines a condition that each row must satisfy. You cannot refer to the CURRVAL, NEXTVAL, and other pseudocolumns in a CHECK constraint. You can use SYSDATE, USER, and other functions while defining a CHECK constraint. You can have multiple CHECK constraints in a column. The following example defines a CHECK constraint on the salary column, which sets a minimum salary for employees:

```sql
CREATE TABLE employee
(emp_id NUMBER(4),
fname VARCHAR2(30),
...
salary NUMBER(7,2) CONSTRAINT sal_chk CHECK (salary >= 500),
...);
```
You can define the CHECK constraint either at the table or column level.

A complete example

The following example shows how to create all types of constraint in a table:

```
CREATE TABLE employee
(emp_id NUMBER(4) PRIMARY KEY,
 fname VARCHAR2(20) NOT NULL,
 lname VARCHAR2(30),
 title VARCHAR2(20) NOT NULL,
 salary NUMBER(7,2),
 mgr_id NUMBER(4),
 commission NUMBER(5,2),
 joindate DATE DEFAULT SYSDATE,
 dept_id NUMBER (4) CONSTRAINT dep_fk REFERENCES department(dept_id),
 CONSTRAINT sal_chk CHECK (salary >= 500));
```

In the above example, the check constraint on the Salary column is defined at the table level and the rest of the constraints are defined at the column level along with column definitions.

You can retrieve information about the constraints that you create in a table by querying the data dictionary view USER_CONSTRAINTS. You can query the data dictionary to confirm the creation of the constraints. The data dictionary view USER_CONS_COLUMNS displays column names along with the name of constraints. The following SELECT statements retrieve information about constraints:

```
SELECT table_name, constraint_name, column_name
FROM user_cons_columns
WHERE table_name = 'EMPLOYEE';
```

The output will be:

```
<table>
<thead>
<tr>
<th>TABLE_NAME</th>
<th>CONSTRAINT_NAME</th>
<th>COLUMN_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEE</td>
<td>DEP FK</td>
<td>DEPT_ID</td>
</tr>
<tr>
<td>EMPLOYEE</td>
<td>SYS_C003129</td>
<td>EMP_ID</td>
</tr>
<tr>
<td>EMPLOYEE</td>
<td>SYS_C003219</td>
<td>FNAME</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```
The ALTER TABLE...ADD CONSTRAINT Statement

You can add a constraint to an existing table using the ALTER TABLE statement. You can create constraints either while creating the table or after creating the table. You have seen how to create constraints while creating a table. You can create constraints after creating the table using the ALTER TABLE...ADD CONSTRAINT statement. The following example shows how to add a constraint to an existing table:

```sql
ALTER TABLE employee
ADD CONSTRAINT mgr_fk FOREIGN KEY (mgr_id) REFERENCES employee(emp_id);
```

The above example defines a FOREIGN KEY on the MGR_ID column of the EMPLOYEE table, which references the EMP_ID column of the same table.

Any constraint created using the ALTER TABLE....ADD CONSTRAINT statement is created at the table level. As a result, you cannot add a NOT NULL constraint using the ALTER TABLE...ADD CONSTRAINT statement because you cannot define a NOT NULL constraint at the table level. You can define a NOT NULL constraint on a column of an existing table using the ALTER TABLE...MODIFY statement, as shown in the following example:

```sql
ALTER TABLE department
MODIFY (name CONSTRAINT name_nn NOT NULL);
```

All constraints are stored in the data dictionary. They are easy to refer to if you provide meaningful names to them, as done in the above example, where you have named the NOT NULL constraint on the Name column of the DEPARTMENT table as NAME_NN. If you do not provide a name to the constraint, the Oracle server takes the responsibility of assigning a name to the constraint using the SYS_Cn format, where n is the unique sequence number generated for each constraint.

**Disable or Enable Constraints**

When you create any constraint, by default, the constraint is enabled or activated. You can explicitly deactivate or disable a constraint using the DISABLE clause in the ALTER TABLE statement. The following example disables the PRIMARY KEY of the Department table:

```sql
ALTER TABLE department
DISABLE CONSTRAINT dep_id_pk;
```

You can use the CASCADE option along with the DISABLE clause to disable dependent integrity constraints. When you disable a PRIMARY KEY or a UNIQUE constraint, the unique index is automatically dropped.

You can explicitly enable a constraint using the ENABLE clause in the ALTER TABLE statement. The ENABLE clause has two options associated with it, VALIDATE and NOVALIDATE. The VALIDATE option, which is the default option, compiles the old data with the constraint. If any row in the table violates the constraint, the constraint remains disabled and the Oracle server returns an error.
The NOVALIDATE option specifies that the new data would be compiled with the constraint. This clause does not ensure that the existing data compiles with the constraint. You can enable a constraint in the following manner:

```
ALTER TABLE employee
ENABLE CONSTRAINT emp_id_pk;
```

The above example uses the VALIDATE option, which also compiles the old data. The following example uses the NOVALIDATE option to enable the check constraint on the SALARY column of the EMPLOYEE table:

```
ALTER TABLE employee
ENABLE NOVALIDATE CONSTRAINT sal_chk;
```

**Dropping a Column**

You can drop a constraint using the DROP CONSTRAINT clause with the ALTER TABLE statement. You can retrieve the name of the constraint from data dictionary views. For example:

```
ALTER TABLE employee
DROP CONSTRAINT mgr_fk;
```

You can drop the PRIMARY KEY using the PRIMARY KEY keyword. You can use the cascade option to drop any dependent constraint, as follows:

```
ALTER TABLE department
DROP PRIMARY KEY CASCADE;
```
Review Checklist: Including Constraints

- Define a table constraint.
- Know valid Oracle constraints and their uses.
- Identify the characteristics of constraints.
- Identify the proper syntax of constraints.
- Query the data dictionary views, USER_CONSTRAINTS and USER_CONS_COLUMNS, to view details about constraints.
- Add a constraint to an existing table using the ADD and MODIFY clauses in the ALTER TABLE statement.
- Disable a constraint using the ALTER TABLE...DISABLE statement.
- Enable a constraint using the ALTER TABLE...ENABLE statement.
Creating Views
Describe a View

Scope

Define a simple and complex view. Identify the uses of a view.

Focused Explanation

A view is a logical table based on a table or another view. A view does not have data of its own. It is like a window to a table through which you can view and change the data of the table. The tables on which a view is based are called base tables. A view is stored in the form of a SELECT statement in the data dictionary.

Views have many advantages and are used to:

- Restrict access to data by selectively displaying columns from a table.
- Create simple queries to retrieve the results of complicated queries.
- Provide data independence for application programs.
- Provide users with access to data based on their requirements.

A view is a SELECT statement based on one or more tables or views. It selectively displays columns from a table, restricting the user’s access to that base table. Views make complex queries simpler. For example, you can use views to query data from multiple tables without knowing how to write a join statement. All operations performed on a view affect the base table. You use views in the same manner as you use tables. You can query, insert into, update, delete from views, just as you perform the same operations on tables.

Views are powerful because they provide different representations of data to different users based on their requirements. You can have multiple views based on the same table. You can create a view in your own schema only if you have the CREATE VIEW system privilege.

Views are of two types, simple and complex:

- Simple view: Retrieves data from only one table and contains no functions or group data. You can perform DML operations through a simple view.
- Complex view: Retrieves data from multiple tables or contains functions or group data. A complex view does not always allow you to perform DML operations.
Create, Alter the Definition, and Drop a View

Scope

Identify the syntax of CREATE VIEW to create a view. Modify a view to include an additional column using the CREATE OR REPLACE VIEW statement. Remove a view. Understand the purpose of the FORCE option when creating a view. Know the characteristics of a view.

Focused Explanation

You can create a view based on other tables or views. You can create a view by embedding a SELECT statement within the CREATE VIEW statement. The general syntax to create a view is:

```
CREATE [OR REPLACE] [FORCE | NO FORCE] VIEW view_name [(alias[, alias]...)]
AS subquery
[WITH CHECK OPTION [CONSTRAINT constraint_name]]
[WITH READ ONLY [CONSTRAINT constraint_name]];
```

In the above syntax, the OR REPLACE option recreates the view if the view already exists; FORCE creates a view irrespective of whether the base table exists or not; NO FORCE creates the view only if the base table exists, it is the default option; view_name is the name of the view; alias specifies the name of the columns or expressions selected by the view’s query; subquery is the SELECT statement; WITH CHECK OPTION specifies a restriction for inserts and updates through views based on the WHERE clause of the SELECT statement of the view; constraint_name is the name of the constraint; WITH READ ONLY specifies that no DML operations can be performed on this view.

Creating Simple Views

You create a simple view based on only one table. In other words, the SELECT statement of the view retrieves data from only one table. For example, you create a view, EMP_VU, based on the EMPLOYEE table that contains the EMP_ID with the alias EMPNO, FIRST_NAME with an alias name ENAME, the annual salary values with an alias name ANNSAL, and DEPT_ID with an alias name DEPTNO.

```
CREATE VIEW emp_vu (empno, ename, annsal, deptno)
AS SELECT emp_id, fname, salary * 12, dept_id FROM employee;
```
You can query the view as you query the table. You can describe the view in the same manner as you describe a table using the iSQL*Plus DESCRIBE command.

```
DESCRIBE emp_vu
```

The output will be:

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPNO</td>
<td>NOT NULL</td>
<td>NUMBER (4)</td>
</tr>
<tr>
<td>ENAME</td>
<td></td>
<td>CHAR (10)</td>
</tr>
<tr>
<td>ANNSAL</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>DEPTNO</td>
<td></td>
<td>NUMBER (4)</td>
</tr>
</tbody>
</table>

You can create a view based on a SELECT statement with a WHERE condition. You can modify the previous example to include a WHERE condition, as follows:

```
CREATE VIEW emp_vu (empno, ename, annsal, deptno)
AS SELECT emp_id, fname, salary * 12, dept_id
FROM employee
WHERE dept_id = 1001;
```

**Creating Complex Views**

A complex view contains group functions or displays data from multiple tables. The following example creates a complex view, which retrieves data from multiple tables and includes a group function:

```
CREATE VIEW dep_comp_vu (dname, minsal, maxsal, total)
AS SELECT d.name, MIN(e.salary), MAX(e.salary), SUM(salary)
FROM department d, employee e
WHERE e.dept_id = d.dept_id
GROUP BY d.name;
```

The above example creates a complex view that retrieves data from the Employee and Department tables and computes the group functions, MAX, MIN, and SUM, in the Salary column of the Employee table. The number of alias names listed in the CREATE VIEW statement must match the number of columns selected in the SELECT list.
Modifying a View

You can modify a view using the CREATE OR REPLACE statement. You need not drop and recreate the view to make the modifications. The CREATE OR REPLACE syntax replaces the old definition of the view with the new definition. For example:

```sql
CREATE OR REPLACE VIEW emp_vu (empno, ename, annsal, job, deptno)
AS SELECT emp_id, fname, salary * 12, title, dept_id
FROM employee
WHERE dept_id = 1001;
```

You have modified the EMP_VU by adding the TITLE column to its definition in the above example. With the OR REPLACE option, a view is created even if a view already exists by the same name, replacing the old version of the view. In other words, you can alter a view without dropping, recreating or reassigning object privileges.

Removing a View

You can remove a view from the database using the DROP VIEW statement, as follows:

```sql
DROP VIEW dept_vu;
```

The DROP VIEW statement removes the view from the database without deleting data because a view is based on underlying tables in the database. Views or other applications based on views that have been dropped become invalid.

The FORCE Option

You can create a view regardless of whether or not the base table exists in the database using the FORCE option in the CREATE VIEW statement, as follows:

```sql
CREATE OR REPLACE FORCE VIEW sal_for_vu
AS SELECT losal, hisal
FROM grade_tab;
```

In the above example, the GRADE_TAB table does not exist in the database. In spite of this, the view is created. You cannot execute a view created using the FORCE option because the base table does not exist. You can use the FORCE option to create a view that refers to a nonexistent table or an invalid column or if the owner of the view does not have the required privileges. This type of view is called View with Errors. The default option is NO FORCE, which specifies that a view be created only if base tables exist or the owner has the required privilege.
Characteristics of a View

A view is always defined by the SELECT statement that retrieves data from base table(s). The characteristic features of a view are:

- The subquery on which a view is based can contain complex SELECT statements, such as join conditions, groups, and subqueries.
- The subquery of the view can contain an ORDER BY clause.
- You can include a GROUP BY clause in the subquery of a view and use group functions.
- You can have a read-only privilege over a view using the WITH READ ONLY option.
- If you do not provide a name to the constraint in the WITH CHECK OPTION and the WITH READ ONLY option, the Oracle server assigns a default name in the format SYS_Cn.
Retrieve Data through a View

Scope

Use the USER_VIEWS data dictionary view to view details about the view you have created. Identify valid
SELECT statements that query a view. Understand the requirements to query a view.

Focused Explanation

Once you have created a view, you can query the data dictionary view USER_VIEWS to retrieve
information about your view. You can also retrieve the text of the SELECT statement that constitutes your
view from the USER_VIEWS data dictionary, as follows:

```
SELECT view_name, text
FROM user_views
WHERE view_name = 'EMP_VU';
```

The above statement retrieves the view definition from the data dictionary.

Querying a View

You can query a view in the same manner as you query a table. When you query a view, the Oracle
server retrieves the view definition from the data dictionary. It then checks for privileges to the view base
table(s). The data is retrieved from base tables. The following statement retrieves data from the base
table through the view:

```
SELECT * FROM emp_vu;
```

```
EMP_VU

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>ANNSAL</th>
<th>JOB</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>3439</td>
<td>JOHN</td>
<td>60000</td>
<td>PRESIDENT</td>
<td>1001</td>
</tr>
<tr>
<td>3382</td>
<td>ROGER</td>
<td>29400</td>
<td>MANAGER</td>
<td>1001</td>
</tr>
</tbody>
</table>

The above statement retrieves data from the base table. When you execute the above statement, the
Oracle server internally issues the following SELECT statement on the base table to achieve the above
output:

```
SELECT employee_id AS empno, fname AS ename, salary * 12 AS annsal,
title AS job, department_id AS Deptno
FROM employee
WHERE department_id = 1001;
```
When you execute a SELECT statement that retrieves all columns from the view, the Oracle server executes the SELECT statement that constitutes the view. If you try to execute a SELECT that has a column listed in its SELECT list, which is not a part of the view definition, but a part of the base table definition, it results in the following error:

```
SELECT empno, ename, salary, deptno
FROM emp_vu;
```

```
ERROR at line 1:
ORA-00904: invalid column name
```

In the above example, the SALARY column is a column that exists in the base table but is not a part of the view definition. EMP_VU consists of EMPLOYEE_ID, FNAME, SALARY*12, TITLE, and DEPARTMENT_ID with alias names as EMPNO, ENAME, ANNSAL, JOB, and DEPTNO, respectively. The view does not have a SALARY column in its definition. In addition, when you refer to any column of the base table through the view, you must refer to that column by the name specified in the view definition. For example, the following statement results in an error because EMPLOYEE_ID is the name of the base table column but the name of that column in the view definition is EMPNO.

```
SELECT employee_id, ename, job FROM emp_vu;
```

```
ERROR at line 1:
ORA-00904: invalid column name
```

The EMPLOYEE_ID column is the base table column referred to as EMPNO in the view. As a result, when you query any table through its view, you must refer to the columns according to the view definition. If your view has a WHERE condition in its definition, ensure that while querying your base table through the view, you must not provide a WHERE condition that violates the WHERE clause of the view definition. In the following example, the query does not retrieve any rows in the result set because the view retrieves data about employees that belong to department 1001. The following query tries to retrieve data about employees that work in department 1003:

```
SELECT *
FROM emp_vu
WHERE deptno = 1003;
```

```
no rows selected
```

As a result, while retrieving data from the base table through a view, you must ensure that your SELECT statement does not violate the view definition. You can query a view only if you have a SELECT privilege over that view.
You can create a view that retrieves data from multiple base tables. This view is called a complex view. For example, the following view retrieves data from the EMPLOYEE and DEPARTMENT tables:

```
CREATE OR REPLACE VIEW dep_empl
AS
SELECT d.location loc, e.title job, e.fname name, e.salary sal,
     e.mgr_id mgr, e.joindate joindate
FROM employee e, department d
WHERE e.dept_id = d.dept_id;
```

You obtain the following output when you describe the above view:

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td></td>
<td>CHAR (13)</td>
</tr>
<tr>
<td>JOB</td>
<td></td>
<td>CHAR (9)</td>
</tr>
<tr>
<td>NAME</td>
<td></td>
<td>CHAR (10)</td>
</tr>
<tr>
<td>SAL</td>
<td></td>
<td>NUMBER (7,2)</td>
</tr>
<tr>
<td>MGR</td>
<td></td>
<td>NUMBER (4)</td>
</tr>
<tr>
<td>JOINDATE</td>
<td></td>
<td>DATE</td>
</tr>
</tbody>
</table>

You can use group functions, the GROUP BY clause, and the HAVING clause in the SELECT statement that queries your view. For example:

```
SELECT loc, job, SUM(sal) sum, COUNT(*) total
FROM dep_empl
GROUP BY loc, job
HAVING SUM(sal) > 2000;
```

The output is:

```
DEP_EMPL

<table>
<thead>
<tr>
<th>LOC</th>
<th>JOB</th>
<th>SUM</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DETROIT</td>
<td>MANAGER</td>
<td>2850</td>
<td>1</td>
</tr>
<tr>
<td>DETROIT</td>
<td>SALESMAN</td>
<td>5600</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```
Insert, Update, and Delete Data through a View

Scope

Create views that allow the user to insert, delete, and modify data through the view. Understand the effects of the WITH CHECK OPTION clause. Know the restrictions in a view created with the WITH READ ONLY option.

Focused Explanation

You can perform DML operations on simple views. Any DML operation performed on a view, affects the base table. A simple view allows you to perform a DML operation on it. You can perform DML operations on data through a view if the DMLs follow certain rules.

Rules for Performing DELETE Operations on a View

You cannot remove a row from a view if the view contains any one of the following:

- Group functions
- A GROUP BY clause
- The DISTINCT clause
- The pseudocolumn ROWNUM keyword
- Multiple base tables

In other words, a complex view does not allow you to perform DML operations. You cannot delete a row in a view, which retrieves data from multiple tables. It results in the following error:

```
DELETE FROM dep_empl;
DELETE FROM dep_empl
  *
ERROR at line 1:
ORA-01752: cannot delete from view without exactly one key-preserved table
```
**Rules for Performing UPDATE Operations on a View**

You cannot update a row in a view if the view contains any one of the following:

- Group functions
- A GROUP BY clause
- The DISTINCT clause
- The pseudocolumn ROWNUM keyword
  - Columns defined by expressions
  - Multiple base tables

You cannot modify a row in a view if the view contains any one of the conditions listed above. You cannot update a view if your view has a column defined by an expression, such as `SALARY * 12`. For example, the following statement results in an error:

```
UPDATE EMP_VU SET annsal = 50000;
UPDATE EMP_VU SET annsal = 50000;
```

```
* ERROR at line 1:
ORA-01733: virtual column not allowed here
```

**Rules for Performing INSERT Operations on a View**

You cannot insert a row in a view if the view contains any one of the following:

- Group functions
- A GROUP BY clause
- The DISTINCT clause
- The pseudocolumn ROWNUM keyword
- Columns defined by expression
- Multiple base tables
- NOT NULL columns in the base table, which are not a part of the view’s definition
You cannot perform insert operations on a view, which has multiple base tables because you cannot insert values in more than one table at a time. You cannot insert a row in a view that does not have the NOT NULL column of the base table in its definition. For example, you create a view Emp_vu1, based on the EMPLOYEE table, which has the EMPLOYEE_ID, FNAME, SALARY, and TITLE columns in its definition. The view does not contain the DEPT_ID column in its definition, which is a NOT NULL column in the base table, EMPLOYEE. If you try to perform insertion in the EMP_VU1 view, it results in the following error:

```
INSERT INTO emp_vu1
VALUES (3221, 'ANDREWS', 3500, 'MANAGER');

INSERT INTO emp_vu1
*
ERROR at line 1:
ORA-01400: cannot insert NULL into ("SCOTT"."EMPLOYEE"."DEPT_ID")
```

When you execute the above statement, the Oracle server internally issues the following insert statement on the base table:

```
INSERT INTO employee (emp_id, fname, lname, title, mgr_id, joindate, salary, commission, dept_id)
VALUES (3221,'ANDREWS',null,'MANAGER',null,null,3500,null,null);
```

In the above statement, a null value is inserted in the DEPT_ID column, which is a NOT NULL column in the base table. This is not allowed. As a result, you cannot insert rows in a view if your view does not include the NOT NULL column of the underlying base table.

**The WITH CHECK OPTION Clause**

You can create views with a SELECT statement and a WHERE condition. For example, you create a view, EMP_VU2, with the following definition:

```
CREATE OR REPLACE VIEW emp_vu2
AS
SELECT emp_id, fname, salary, dept_id
FROM employee
WHERE dept_id = 1001;
```

The above view consists of employees who work in the 1001 department. You must not perform this type of insert and update through the view because this results in rows that the view subquery cannot select. For example, if you update DEPT_ID to 1002 through EMP_VU, the update would be done on the underlying EMPLOYEE table. Your view cannot select any row from the base table because the SELECT statement of the view retrieves rows where DEPT_ID = 1001. You have already updated those rows to DEPT_ID = 1002. As a result, there are no rows in the base table that match the WHERE condition of the view’s SELECT statement. In other words, you must ensure that you do not perform any inserts or
updates through the view on the base table, which would result in rows that cannot be selected by the view subquery. You can do this using the WITH CHECK OPTION clause. You can modify the creation of the EMP_VU2 view, as follows:

```sql
CREATE OR REPLACE VIEW emp_vu2 AS
SELECT emp_id, fname, salary, dept_id
FROM employee
WHERE dept_id = 1001
WITH CHECK OPTION;
```

The above clause specifies that the INSERTs and UPDATEs performed through the view cannot create rows that the view cannot select, enforcing data validation and integrity constraints on data being inserted or updated. As a result, if an attempt is made to perform an update or insert, which results in rows that cannot be selected by the view, you will encounter the following error:

```
UPDATE emp_vu2
SET dept_id = 10002;
```

```
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

**WITH READ ONLY Option**

You can ensure that no DML operations are performed through your view by adding the WITH READ ONLY clause to your view definition. You can create a view with the WITH READ ONLY option, as follows:

```sql
CREATE OR REPLACE VIEW emp_vu3 AS
SELECT emp_id, fname, salary, title
FROM employee
WITH READ ONLY;
```

The above statement creates a view that provides READ ONLY access to the view to users. In other words, you cannot perform DML operations on this view. As a result, the following statement would result in an error:

```
DELETE FROM emp_vu3;
```

```
ERROR at line 1:
ORA-01752: cannot delete from view without exactly one key-preserved table
```
Create and Use an Inline View

Scope

Define an inline view. Identify the correct syntax to create SELECT statements with inline views.

Focused Explanation

You have used subqueries in the WHERE clause of a SELECT statement, in the SET clause of an UPDATE statement, in the INTO clause of an INSERT statement, and in the WHERE clause of a DELETE statement. Apart from this, you can use a subquery in the FROM clause of a SELECT statement. An inline view is a subquery with an alias name placed within the FROM clause of a SELECT statement. The subquery acts as a data source for the main query. An inline view is not a schema object.

For example:

```sql
SELECT e.fname fname, e.salary salary, e.dept_id dept_id, a.salavg salavg
FROM employee e, (SELECT dept_id, AVG(salary) salavg
    FROM employee
    GROUP BY dept_id) a
WHERE e.dept_id = a.dept_id
AND e.salary > a.salavg;
```

The above statement retrieves rows that contain details of employees who earn salaries more than the average salary of their department. In the above example, the inline view retrieves details of all department numbers and the average salaries of each department from the EMPLOYEE table. The WHERE condition of the main query, WHERE e.dept_id = a.dept_id AND e.salary > a.salavg, displays details of employees who earn salaries more than the average salary of their department.

The output will be:

<table>
<thead>
<tr>
<th>FNAME</th>
<th>SALARY</th>
<th>DEPT_ID</th>
<th>SALAVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOHN</td>
<td>5000</td>
<td>1001</td>
<td>2916.66667</td>
</tr>
<tr>
<td>PATRICK</td>
<td>2975</td>
<td>1002</td>
<td>2175</td>
</tr>
<tr>
<td>STEVE</td>
<td>3000</td>
<td>1002</td>
<td>2175</td>
</tr>
</tbody>
</table>

An inline view is a SELECT statement substituted for a table in the main SELECT statement. You always refer to the inline view with its alias name.
Perform Top ‘N’ Analysis

Scope

Write SELECT statements that use Top ‘N’ Analysis. Know the requirements to use Top ‘N’ Analysis. Identify scenarios that require Top ‘N’ Analysis.

Focused Explanation

Top ‘N’ Analysis is a scenario where you need to display only the n-top or the n-bottom records from a table based on a condition. Top-n queries ask for the n largest or n smallest values of a column. Both the largest and smallest values are considered top-n queries.

The general syntax for the high-level structure of a top-n analysis is:

```
SELECT ROWNUM, [column_list]
FROM (SELECT [column_list]
FROM table
ORDER BY Top-N_col)
WHERE ROWNUM <= N;
```

A top-n query must consist of the following:

- A subquery in the FROM clause or an inline view to provide a sorted list of data. The inline view includes an ORDER BY clause to sort the ranking in the desired order. You use DESC to retrieve the largest values.

- An outer query to limit the number of rows in the result set. The outer query includes the following:
  - The ROWNUM pseudocolumn, which assigns a row number to each row, starting with 1.
  - A WHERE clause that specifies the n rows to be returned. You must use the < or <= operators in the outer WHERE clause.
The following example displays the top three earners’ names and salaries. These employees belong to department 1002 in the EMPLOYEE table:

```sql
SELECT ROWNUM, fname, salary
FROM (SELECT fname, salary
      FROM employee
      WHERE dept_id=1002
      ORDER BY salary DESC)
WHERE ROWNUM <=3;
```

In the above example, the inline view retrieves the first name and salary of employees working in department 1002 in descending order of their salary. The WHERE condition of the outer query specifies that the result set would return only the first three rows.

For example, you can use the top-n analysis for the following types of queries:

- Top four earners in the company
- Two recently hired salesmen in the company
- The three most senior employees of the company

Another example of the top-n analysis is:

```sql
SELECT ROWNUM, fname, joindate
FROM (SELECT fname, joindate
      FROM employee
      WHERE title='SALESMAN'
      ORDER BY joindate DESC)
WHERE ROWNUM <= 2;
```

The above query displays the two most recently hired salesmen in the company.
Review Checklist: Creating Views

- Describe different types of views.
- Create a view using the CREATE VIEW statement.
- Modify a view using the CREATE OR REPLACE VIEW statement.
- Drop a view using the DROP VIEW statement.
- Create a view without the base table using the FORCE option.
- Know the characteristics of a view.
- Query the USER_VIEWS data dictionary view to retrieve information about views.
- Identify valid SELECT statements that query a view.
- Create views that allow inserts, updates, and deletes through the view.
- Understand the effects of WITH CHECK OPTION.
- Use the WITH READ ONLY option to restrict the user from performing DML operations through views.
- Create and use inline views.
- Perform Top ‘N’ Analysis using inline views.
Creating Other Database Objects
Create, Maintain, and Use Sequences

Scope

Identify the characteristics of a sequence. Use the CREATE SEQUENCE statement to create a sequence. Use the CURRVAL and NEXTVAL pseudocolumns to populate column values. Modify a sequence and know the limitations while modifying a sequence. Remove a sequence using the DROP SEQUENCE statement.

Focused Explanation

You require the use of unique numbers for PRIMARY KEY columns. You can achieve this by either building code to generate unique values or using a sequence to generate unique numbers. A sequence is a user-defined database object that generates unique values automatically and can be shared by multiple users. Sequences are used to populate PRIMARY KEY columns. Sequences are created by users but are generated and incremented by internal Oracle routines.

The general syntax to create a sequence is:

```sql
CREATE SEQUENCE sequence_name
[INCREMENT BY n]
[START WITH n]
[{MAXVALUE n | NOMAXVALUE}]
[{MINVALUE n | NOMINVALUE}]
[{CYCLE | NOCYCLE}]
[{CACHE n | NOCACHE}];
```

In the above syntax:

- `sequence_name`: Is the name of the sequence generator.
- `INCREMENT BY n`: Defines the interval of n between sequence numbers. The default value of n is 1.
- `START WITH`: Specifies the start value of the sequence. The default value of n is 1.
- `MAXVALUE n`: Defines the maximum value that a sequence can generate.
- `NOMAXVALUE`: Defines a maximum value of 10^27 for an ascending sequence and -1 for a descending sequence. This is the default option.
- `MINVALUE n`: Specifies the minimum value of a sequence.
- `NOMINVALUE`: Specifies a minimum value of 1 for an ascending sequence and -(10^26) for a descending sequence. This is the default option.
- `CYCLE | NOCYCLE`: Defines whether a sequence continues to generate value after reaching its maximum or minimum value. The default option is NOCYCLE.
- `CACHE | NOCACHE`: Specifies how many values the Oracle server should preallocate and keep in memory. The default value for n is 20.
The following example creates a sequence, called DEPT_ID_SEQ, which populates the DEPT_ID value in the DEPARTMENT table:

```sql
CREATE SEQUENCE dept_id_seq
INCREMENT BY 1
START WITH 1001
MAXVALUE 1010
NOCYCLE
NOCACHE;
```

The above sequence starts with a value of 1001, increments the sequence by 1, generates up to a value of 1010, does not have a MINVALUE, does not CYCLE, and does not allow caching. You should not use the CYCLE option if you need to create a sequence to populate a PRIMARY KEY column because the CYCLE option allows the sequence to continue generating values after reaching its maximum value. As a result, a sequence with the CYCLE option tends to generate duplicate values that cannot be inserted in a PRIMARY KEY column. You can confirm sequence creation by querying the USER_SEQUENCES data dictionary view, as follows:

```sql
SELECT sequence_name, min_value, max_value, increment_by, last_number
FROM user_sequences
WHERE sequence_name = 'DEPT_ID_SEQ';
```

In the above statement, LAST_NUMBER denotes the next available sequence number if NOCACHE is specified.

A sequence is not associated with any table at the time of its creation. A sequence is an independent object that can be used anywhere.

**NEXTVAL and CURRVAL Pseudocolumns**

A sequence generates sequential numbers for use in tables. You can reference sequence number values using the NEXTVAL and CURRVAL pseudocolumns. The NEXTVAL pseudocolumn generates successive sequence numbers from a specified sequence. You can use the pseudocolumn as sequence_name.NEXTVAL. When you use sequence_name.NEXTVAL, a new sequence number is generated and the current sequence number is placed in CURRVAL, which is also referenced as sequence_name.CURRVAL.

You can use CURRVAL and NEXTVAL in the SELECT list of a SELECT statement only if the SELECT statement is not a part of a subquery, in the SELECT list of a subquery in an INSERT statement, in the VALUES clause of an INSERT statement, and in the SET clause of an UPDATE statement. You cannot
use pseudocolumns in the SELECT list of a view, a SELECT statement with the DISTINCT keyword; GROUP BY, HAVING, or ORDER BY clauses, a subquery in a SELECT, INSERT, UPDATE, or DELETE statement, and the DEFAULT clause in the CREATE TABLE or ALTER TABLE statement. You can use the sequence to insert values in a table column, as follows:

```
INSERT INTO department (dept_id, name, location)
VALUES (dep_id_seq.nextval, 'SUPPORT', 'DALLAS');
```

The above statement inserts the value generated by the DEPT_ID_SEQ sequence in the DEPT_ID column of the DEPARTMENT table.

You use the cache option to have faster access to sequence numbers. The cache is populated the first time you refer to the sequence. For every request you make for the next sequence, a value is retrieved from the cached sequence. Although the sequence generates sequence numbers without gaps, you can have gaps in your sequence value due to other factors. If you rollback a statement containing a sequence value, that value is lost. You can also have gaps created in your sequence by a system crash. All cached sequence values are lost in case of system crashes.
Create and Maintain Indexes

Scope

Identify constraint columns that are automatically indexed. Create an index using the CREATE INDEX syntax. Know the characteristics of indexes. Identify the conditions to create an index. Remove an index.

Focused Explanation

An index is a database object that can speed up the retrieval of rows by using a pointer. An index provides direct and faster access to the rows of a table. If you do not have an index in a table, a full table scan occurs. The Oracle server automatically uses and maintains indexes. The index reduces the requirement of disk I/O using an indexed path to locate data quickly. You can create indexes automatically or manually. A unique index is automatically created when you create a PRIMARY KEY or a UNIQUE column. You can manually create nonunique indexes on columns to speed up the retrieval of rows by using the CREATE INDEX statement, as follows:

```
CREATE INDEX index_name
ON table_name (col1 [, col2]...);
```

In the above syntax, index_name is the name of the index, table_name is the name of the table for which the index would be created, and col1 is the name of the column on which the index would be created. The following example creates an index on the DEPT_ID column of the EMPLOYEE table:

```
CREATE INDEX Dep_id_idx
ON employee (dept_id);
```

You can create an index in your schema only if you have the CREATE INDEX privilege. Because indexes speed up the retrieval of rows, more indexes does not mean faster access. Any commit of DML operations on a table with indexes requires the update of the indexes. The more indexes you have in a table, the more effort is required from the Oracle server to update the indexes after any DML operation on that table. Therefore, you should create indexes based on the following conditions:

- If the column contains a wide range of values.
- If the column has a large number of null values.
- One or more columns are frequently used in a join or WHERE condition.
- The table is large enough and most queries retrieve data less than 2 to 4% of the rows.
You should create an index only if the above conditions are true. You must not create indexes if:

- The table is small.
- Columns are not frequently used in the query condition.
- Most queries retrieve data more than 2 to 4% of the rows in a table.
- The table is frequently updated.
- The indexed column is referenced frequently as a part of an expression.

You can confirm the creation of indexes by querying the `USER_INDEXES` and `USER_IND_COLUMNS` data dictionary views, as follows:

```sql
SELECT ui.index_name, ui.uniqueness, ui.table_name, uic.column_name,
       uic.column_position
FROM user_indexes ui, user_ind_columns uic
WHERE ui.index_name=uic.index_name
AND ui.table_name = 'EMPLOYEE';
```

You create a nonunique index, by default, when you create an index manually. You can create a unique index manually using the `UNIQUE` keyword in the `CREATE INDEX` statement.
Create Private and Public Synonyms

Scope

Identify reasons to create synonyms. Know the proper syntax to create private and public synonyms. Know the effects of creating private and public synonyms. Drop a synonym.

Focused Explanation

A synonym is a database object that provides a better name to a table, view, sequence, or package. Synonyms are used for security because they hide the identity of underlying objects. Synonyms provide location transparency for remote objects in a distributed database environment. There are two types of synonyms, private and public. A private synonym is a synonym present in the schema of a particular user who has control over the availability of the synonym to other users. A public synonym is a synonym accessible to all users. You must have the CREATE SYNONYM privilege to create a private synonym and the CREATE PUBLIC SYNONYM privilege to create a public synonym. The general syntax to create a synonym is:

```
CREATE [PUBLIC] SYNONYM synonym FOR object;
```

In the above syntax, synonym is the name of the synonym and object is the name of the object for which the synonym is created. In the following example, a private synonym, called DEP_JOB, is created for the DEPT_JOB_VIEW view:

```
CREATE SYNONYM dep_job FOR dept_job_view;
```

The above synonym is a private synonym, which is accessible only to the owner of the synonym. You can create a public synonym, as follows:

```
CREATE PUBLIC SYNONYM emp_sal FOR emp_sal_vu;
```

A public synonym is accessible to all users. To refer to any object owned by another user, you must prefix the object name with the owner’s name followed by a period. Therefore, you can create a synonym to eliminate the need to qualify an object name with the schema and provide an alternative name for the object. As a result, synonyms can be used to simplify SQL statements. In the following example, you create a synonym for the DEP_TAB table that belongs to user JAMES:

```
CREATE SYNONYM dep FOR james.dept_tab;
```

You can confirm the creation of your synonym by querying the USER_SYNONYMS data dictionary view.
Review Checklist: Creating Other Database Objects

- Identify the characteristics of a sequence.
- Create sequences using the CREATE SEQUENCE statement.
- Use the NEXTVAL and CURRVAL pseudocolumns to populate column values.
- Query the data dictionary view USER_SEQUENCES to confirm the creation of the sequence.
- Create an index using the CREATE INDEX statement.
- Know the two types of indexes, unique and nonunique.
- Know the characteristics of an index.
- Query the USER_INDEXES and USER_IND_COLUMNS data dictionary views to confirm the creation of the index.
- Create private and public synonyms using the CREATE SYNONYM statement.
- Query the USER_SYNONYMS data dictionary view to confirm the creation of the synonym.
Controlling User Access
Create Users

Scope

Know the default privileges. Identify the proper syntax to create a user. Identify the proper syntax to change a user’s password.

Focused Explanation

Multiple users can access a database. In this case, you need to maintain the security and use of database access. Oracle allows you to control database access, allow access to certain objects in the database, confirm granted and revoked privileges from the Oracle data dictionary, and create synonyms for database objects.

Database security is classified into two categories, system security and data security. System security includes the access and use of databases at system level, such as the login ID, password, and disk space allocated to users. Database security includes the access and use of database objects and the privileges that users can have on the objects.

A schema is a collection of objects that belong to a particular user. A database user owns a schema, which has the same name as the user.

The Database Administrator (DBA) creates users using the CREATE USER statement. The general syntax to create a user is:

```
CREATE USER username
IDENTIFIED BY password;
```

In the above syntax, username is the name of the user and password specifies that the user must log on using the password. In the following example, a user, named JAMES, has been created and provided a password to log on:

```
CREATE USER james
IDENTIFIED BY bond;
```

To maintain database security, the DBA, who is a high-level user capable of granting users access to the database and its objects, grants the privilege to access the database and its objects to the users. Privileges are the right to execute specific SQL statements or to access certain database objects. Privileges are of two types, system and object. Users require system privileges to access the database and object privileges to manipulate database objects.
System Privileges

System privileges are privileges that allow users to access the database. Oracle provides more than a 100 system privileges. The DBA has high-level system privileges to create other users, grant privileges to users, back up tables, and so on. Some of the system privileges are:

<table>
<thead>
<tr>
<th>System Privilege</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE SESSION</td>
<td>Connects to the database</td>
</tr>
<tr>
<td>CREATE USER</td>
<td>Creates a user</td>
</tr>
<tr>
<td>CREATE ROLE</td>
<td>Creates a role</td>
</tr>
<tr>
<td>CREATE TABLE</td>
<td>Creates a table in the user’s schema</td>
</tr>
<tr>
<td>CREATE SEQUENCE</td>
<td>Creates a sequence in the user’s schema</td>
</tr>
<tr>
<td>CREATE VIEW</td>
<td>Creates a view in the user’s schema</td>
</tr>
<tr>
<td>CREATE SYNONYM</td>
<td>Creates a synonym in the user’s schema</td>
</tr>
<tr>
<td>CREATE ANY TABLE</td>
<td>Creates a table in any schema</td>
</tr>
<tr>
<td>ALTER USER</td>
<td>Alters a user</td>
</tr>
<tr>
<td>DROP USER</td>
<td>Drops a user</td>
</tr>
<tr>
<td>DROP ANY ROLE</td>
<td>Drops a role</td>
</tr>
<tr>
<td>DROP ANY TABLE</td>
<td>Drop tables in any schema</td>
</tr>
</tbody>
</table>

The DBA creates a user by using the CREATE USER statement. A user does not have any associated privileges when the user is created. Both system privileges and object privileges must be explicitly granted to the user after creation. After a user is created, the DBA grants privileges to the user. The privileges determine the access of a user to the database. The general syntax to grant system privileges to the user is:

```
GRANT privilege [, privilege...] TO user [, user| role, PUBLIC...];
```

In the above syntax, privilege is the system privilege granted to user user or to role role or to PUBLIC, which means all users.

After you create a user, you can change the password of the user using the ALTER USER statement, as follows:

```
ALTER USER james
IDENTIFIED BY bend;
```

You can execute the above statement only if you have the ALTER USER system privilege. After executing the above statement, the user JAMES would log on using the password BEND and not BOND.
Create Roles to Ease the Setup and Maintenance of the Security Model

Scope

Define roles. Use the CREATE ROLE statement to create roles and the GRANT statement to grant privileges to the created roles. Identify the characteristics and uses of roles. Grant roles to users.

Focused Explanation

The DBA creates users and grants privileges to the users. Different set of privileges can be assigned to different users. Among the different privileges assigned to different users, you may have a set of privileges common for multiple users. There is a simpler way to grant the same set of privileges to different users by means of using roles. A role is a named set of privileges you can grant to users. This method makes it easier to maintain privileges and revoke them from users, if the need arises. You can grant several roles to a users and a role to several users. You can create roles using the CREATE ROLE statement. The general syntax is:

```
CREATE ROLE role_name;
```

In the above syntax, role_name is the name of the role to be created. A role is a set of privileges granted to users for easy maintenance of privileges. The DBA creates the roles and assigns them to users. The following example creates a role, named SUPPORT:

```
CREATE ROLE support;
```

After the role has been created, the DBA has to grant system privileges to the role, as follows:

```
GRANT CREATE SESSION, CREATE TABLE, CREATE USER TO support;
```

The SUPPORT role has been granted the CREATE SESSION, CREATE TABLE, and CREATE USER system privileges. The next step is to assign the roles to users. The following example assigns the SUPPORT role to users JAMES and ALLEN:

```
GRANT support TO james, allen;
```

In other words, users JAMES and ALLEN are granted the CREATE SESSION, CREATE TABLE, and CREATE USER system privileges. Similarly, you can assign multiple roles to a user. The following example grants the SUPPORT and MANAGER roles to user SCOTT:

```
GRANT support, manager TO scott;
```

As a result, user SCOTT would be granted all the privileges assigned to each role. A role groups related sets of privileges to simplify granting privileges to and revoking privileges from users.
Use the GRANT and REVOKE Statements to Grant and Revoke Object Privileges

Scope

Allow users to query tables. Use the USER_COL_PRIVS_RECD and USER_COL_PRIVS_MADE views to display the privileges granted to a user on specific columns. Understand the cascading effect of revoking object privileges from a user. Identify the privileges that can be granted for a table and a view. Use the WITH GRANT OPTION clause.

Focused Explanation

You have been granting system privileges to users and roles. To manipulate database objects, a user must be assigned object privileges. An object privilege is the right to perform a specific action on a particular database object, such as a table, a view, a sequence, or a procedure. Each object has a set of privileges associated, which can be granted to users or roles. The following table describes some of the object privileges:

<table>
<thead>
<tr>
<th>Object Privileges</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER</td>
<td>Changes the definition of the object</td>
</tr>
<tr>
<td>SELECT</td>
<td>Queries the table view</td>
</tr>
<tr>
<td>INSERT</td>
<td>Adds values to the table</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Modifies the data in the table</td>
</tr>
<tr>
<td>DELETE</td>
<td>Removes data from the table</td>
</tr>
<tr>
<td>INDEX</td>
<td>Creates an index on the table</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>Creates a constraint that references the table</td>
</tr>
</tbody>
</table>

Granting Privileges

You can grant object privileges to the user using the GRANT statement. The syntax for the GRANT statement is:

```
GRANT object_priv [(columns)]
ON object
TO {user | role | PUBLIC}
[WITH GRANT OPTION];
```

In the above syntax, object_priv is the object privilege to be granted, columns specifies the columns in the tables or views on which privileges are granted, object is the name of the object on which privileges are granted, PUBLIC specifies grant privileges to all users, WITH GRANT OPTION allows the grantee to grant privileges to other users.
In the following example, the object privilege required to query the EMPLOYEE table is granted to users JAMES and ALLEN:

```
GRANT select ON scott.employee
TO james, allen;
```

You can grant the object privilege on a particular column to the user, as follows:

```
GRANT update(salary)
ON scott.employee
TO james;
```

In the above example above, JAMES is granted the privilege of updating only the SALARY column of the EMPLOYEE table belonging to the SCOTT schema.

You can query the data dictionary views USER_COL_PRIVS_RECD and USER_COL_PRIVS_MADE to display the privileges granted to the user on specific columns of a table and the privileges granted by the user on specific columns of a table.

A user can grant privileges to another user only if the user has been granted privileges using the WITH GRANT OPTION clause. For example, the DBA grants some object privileges to user USER_A in the following manner:

```
GRANT select, update(salary), insert
ON scott.employee
TO user_a
WITH GRANT OPTION;
```

USER_A creates another user, USER_B, to whom USER_A needs to grant object privileges. Because the DBA has granted object privileges to USER_A using the WITH GRANT OPTION clause, USER_A can grant the same privileges to other users, as follows:

```
GRANT select, update(salary)
ON scott.employee
TO user_b;
```

USER_B cannot grant privileges to any other user because USER_A has not provided WITH GRANT OPTION to USER_B. USER_A can grant only those privileges that are granted to USER_A to other users. USER_A cannot grant any other user a privilege not granted to USER_A.
There are different object privileges available for different types of database objects. The following table lists the object privileges available for a table and a view:

<table>
<thead>
<tr>
<th>Object Privilege</th>
<th>Table</th>
<th>View</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SELECT</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DELETE</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>INSERT</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>UPDATE</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>INDEX</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>REFERENCES</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>EXECUTE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Revoking Privileges**

You have, so far, granted privileges to users and roles. You may, at some point in time, realize that users are misusing privileges and you may want to remove the privileges from the users. You can do this using the REVOKE statement. The general syntax to revoke privileges from users or roles is:

```sql
REVOKE {privilege [, privilege...] | ALL}
ON object
FROM {user[, user...]|role|PUBLIC}
[CASCADE CONSTRAINTS];
```

In the above syntax, CASCADE CONSTRAINTS removes any referential integrity constraints made to the object using the REFERENCES privilege. For example;

```sql
REVOKE select, update(salary)
ON scott.employee
FROM user_a;
```

The SELECT object privilege and the UPDATE privilege on the salary column of the EMPLOYEE table that belongs to SCOTT are revoked from USER_A. The REVOKE statement revokes the specified privileges from users that you name in the statement and from users to whom privileges have been granted using WITH GRANT OPTION. Object privileges are revoked in a cascading manner. If the privilege is revoked from you, it is also revoked from anyone you granted the privilege to. As a result, in the above example, when you revoke privileges from USER_A, privileges are also revoked from USER_B to whom USER_A had granted privileges.
Review Checklist: Controlling User Access

- Create users using the CREATE USER statement.
- Identify system privileges and the proper syntax to grant them using the GRANT statement.
- Modify the password of a user using the ALTER USER statement.
- Define roles and create them by using the CREATE ROLE statement.
- Identify the characteristics and uses of roles.
- Grant privileges to roles using the GRANT statement.
- Grant roles to users.
- Allow users to query tables by granting the object privilege to users.
- Query data dictionary views to display the privileges granted to and by the user on specific columns.
- Know the different object privileges available in an Oracle database.
- Identify the proper syntax to revoke object privileges from a user.
- Understand the cascading effect of revoking object privileges from a user.
Test Taking Strategies

The Oracle Certified Associate, Professional, and Master Credentials identify a standard of competence for entry-level and professional job roles that utilize Oracle products. Oracle’s certification program is a recognized credential that signifies a proven level of knowledge and ability. With each level of certification, a higher benchmark of ability is set for greater opportunities and higher pay.

The 1Z0-007 exam is the starting point for many of the current Oracle certification tracks. This exam is un-proctored, meaning you can take the exam at your own location using the Internet. If you do not meet the minimum Internet connection requirements, you can take a proctored exam at an Oracle University Training Center or an Authorized Prometric Testing Center.

To review the Internet Connection requirements for the un-proctored version of this exam:


Oracle Certification Roadmap

The 1Z0-007 Introduction to Oracle9i: SQL exam is the starting point for the Oracle9i Database Administrator and Oracle9i Application Developer tracks. Whether you are seeking an Oracle9i Associate, Professional, or Master credential, you will need to pass the 1Z0-007 exam.

An Oracle candidate should combine training with on-the-job experience. Many of the exam questions are based on real-world scenarios so hands-on experience with the software is vital.

Registering for the Exam

To register for the online exam:

http://oracle.prometric.com/

To register for the proctored version:

Resources

Because the exam is based on the Introduction to Oracle9i: SQL instructor-led training, attending this course is the best preparation. However, if you’re unable to attend this class or don’t have access to the materials, you can use the Oracle9i SQL Reference on the Oracle Technology Network (OTN) to prepare for the exam:

http://download-west.oracle.com/docs/cd/B10501_01/server.920/a96540/toc.htm

Test Day Strategies

The most important test day strategy is be thoroughly prepared for the exam beforehand. You must know the material. Cramming the day of the exam is not a good strategy to use for any type of test, especially certification exams. Oracle highly discourages cheating on the un-proctored version. The consecutive proctored exams require SQL proficiency and cheating on this exam will prevent you from passing the other exams.

Oracle allows you to go back to questions that you previously answered, so manage your time wisely. If a question is requiring too much time to answer, you can always select the best possible response, and then return to the question after answering all of the other questions.

General Tips:

• Schedule your exam only after you are confident that you have mastered the subject matter.
• Schedule your exam for a time of day when you perform at your best.
• Eliminate all distractions from your testing area.
• Allow 2 ½ hours to complete the registration and exam.
• Eat a light meal beforehand.
• Review the question types carefully before starting the actual exam. Be careful not to bypass this option because you are in a hurry to finish the exam.
• Everything we do has time limitations, so don't let the pressure overwhelm you.

Introduction to Oracle9i: SQL-Specific Tips:

• Before starting the exam, flip through your short stack of reserved flash cards, Review Checklists and/or personal study notes to remind yourself about terms, topics, and syntax that are likely to appear on the exam.
• Determine how much time you are allotted to answer each question. Do not spend too much time on a given question during your first pass through the exam. If a question is requiring too much time to answer, select the best possible answer, mark the question for review, and return to the question after answering all of the other questions.
• Remember that if you are interrupted during the online exam, the time clock continues.
• If you are disconnected during your exam, you will be able to resume where you left off.
Test Items

The 1Z0-007 exam contains only multiple-choice items. While knowing the technical content for this exam is the most important thing you can do to pass the exam, understanding the methodology of the question type and following a strategy of how to answer each type can mean the difference between passing and failing. Below we provide you with specific advice for multiple-choice questions.

Multiple-Choice

1. Read each multiple-choice question with the intention of answering the question without the alternatives that follow. Focus on finding an answer without the help of the alternatives. This will increase your concentration and help you read the question more clearly.

2. Use the process of elimination when you do not know the answer for sure. If the question has a single answer, and four options are listed, eliminate two of these options quickly and then make the decision between the two that remain. This increases your probability to 50/50. Another helpful method of elimination is to use a true-false approach where you identify a likely false alternative. Then, you eliminate it. The true-false elimination method is particularly helpful when the question requires more than one answer.

3. When two very similar answers appear, it is likely that one of them is the correct choice. Test writers often disguise the correct option by giving another option that looks very much like the correct one.

You can download a free demo on our website that mimics the types of questions that will appear on the exam. Sample questions do not cover all the content areas on the exam.
YOU’RE ALREADY SMART, WE HELP YOU BE SMARTER.

Thank You and Good Luck on your Exams!

Thank you for purchasing a Self Test Software certification exam preparation product. Self Test helps you figure out what you know and gives you the answers to what you don’t – it’s the best way to prepare for your certification exams.

For your upcoming exam and throughout your IT career, Self Test is the only tool you need to improve your IT IQ and succeed on exam day.

The Self Test Software Team