Oracle Database 12*c*: SQL Workshop II

Student Guide - Volume I

D80194GC11

Edition 1.1

July 2014

D87270

ORACLE

Author

Dimpi Rani Sarmah

Technical Contributors and Reviewers

Nancy Greenberg

Swarnapriya Shridhar

Bryan Roberts

Laszlo Czinkoczki

KimSeong Loh

Brent Dayley

Jim Spiller

Christopher Wensley

Maheshwari Krishnamurthy

Daniel Milne

Michael Almeida

Diganta Choudhury

Manish Pawar

Clair Bennett

Yanti Chang

Joel Goodman

Gerlinde Frenzen

Madhavi Siddireddy

Editors

Raj Kumar

Malavika Jinka

Publishers

Jobi Varghese

Pavithran Adka

Contents

1 Introduction
Lesson Objectives 1-2
Lesson Agenda 1-3
Course Objectives 1-4
Course Prerequisites 1-5
Course Agenda 1-6
Lesson Agenda 1-7
Tables Used in This Course 1-8
Appendixes and Practices Used in This Course 1-10
Development Environments 1-11
Lesson Agenda 1-12
Review of Restricting Data 1-13
Review of Sorting Data 1-14
Review of SQL Functions 1-15
Review of Single-Row Functions 1-16
Review of Types of Group Functions 1-17
Review of Using Subqueries 1-18
Review of Managing Tables Using DML Statements 1-20
Lesson Agenda 1-22
Oracle Database SQL Documentation 1-23
Additional Resources 1-24
Summary 1-25
Practice 1: Overview 1-26

2 Introduction to Data Dictionary Views
Objectives 2-2
Lesson Agenda 2-3
Data Dictionary 2-4
Data Dictionary Structure 2-5
How to Use the Dictionary Views 2-7
USER_OBJECTS and ALL_OBJECTS Views 2-8
USER_OBJECTS View 2-9
Lesson Agenda 2-10
Table Information 2-11
Column Information 2-12

Constraint Information 2-14
USER_CONSTRAINTS: Example 2-15
Querying USER_CONS_COLUMNS 2-16
Lesson Agenda 2-17
Adding Comments to a Table 2-18
Quiz 2-19
Summary 2-20
Practice 2: Overview 2-21

3 Creating Sequences, Synonyms, and Indexes
Objectives 3-2
Lesson Agenda 3-3
Database Objects 3-4
Referencing Another User's Tables 3-5
Sequences 3-6
CREATE SEQUENCE Statement: Syntax 3-7
Creating a Sequence 3-9
NEXTVAL and CURRVAL Pseudocolumns 3-10
Using a Sequence 3-12
SQL Column Defaulting Using a Sequence 3-13
Caching Sequence Values 3-14
Modifying a Sequence 3-15
Guidelines for Modifying a Sequence 3-16
Sequence Information 3-17
Lesson Agenda 3-18
Synonyms 3-19
Creating a Synonym for an Object 3-20
Creating and Removing Synonyms 3-21
Synonym Information 3-22
Lesson Agenda 3-23
Indexes 3-24
How Are Indexes Created? 3-25
Creating an Index 3-26
CREATE INDEX with the CREATE TABLE Statement 3-27
Function-Based Indexes 3-29
Creating Multiple Indexes on the Same Set of Columns 3-30
Example of Creating Multiple Indexes on the Same Set of Columns 3-31
Index Information 3-32
USER_INDEXES: Examples 3-33
Querying USER_IND_COLUMNS 3-34
Removing an Index 3-35

Quiz 3-36
Summary 3-37
Practice 3: Overview 3-38

4 Creating Views
Objectives 4-2
Lesson Agenda 4-3
Database Objects 4-4
What Is a View? 4-5
Advantages of Views 4-6
Simple Views and Complex Views 4-7
Lesson Agenda 4-8
Creating a View 4-9
Retrieving Data from a View 4-12
Modifying a View 4-13
Creating a Complex View 4-14
View Information 4-15
Lesson Agenda 4-16
Rules for Performing DML Operations on a View 4-17
Using the WITH CHECK OPTION Clause 4-20
Denying DML Operations 4-21
Lesson Agenda 4-23
Removing a View 4-24
Quiz 4-25
Summary 4-26
Practice 4: Overview 4-27

5 Managing Schema Objects
Objectives 5-2
Lesson Agenda 5-3
Adding a Constraint Syntax 5-4
Adding a Constraint 5-5
Dropping a Constraint 5-6
Dropping a CONSTRAINT ONLINE 5-7
ON DELETE Clause 5-8
Cascading Constraints 5-9
Renaming Table Columns and Constraints 5-11
Disabling Constraints 5-12
Enabling Constraints 5-13
Constraint States 5-14
Deferring Constraints 5-15

Difference Between INITIALLY DEFERRED and INITIALLY IMMEDIATE 5-16
DROP TABLE … PURGE 5-18
Lesson Agenda 5-19
Temporary Tables 5-20
Creating a Temporary Table 5-21
Lesson Agenda 5-22
External Tables 5-23
Creating a Directory for the External Table 5-24
Creating an External Table 5-26
Creating an External Table by Using ORACLE_LOADER 5-28
Querying External Tables 5-30
Creating an External Table by Using ORACLE_DATAPUMP: Example 5-31
Quiz 5-32
Summary 5-33
Practice 5: Overview 5-34

6 Retrieving Data by Using Subqueries
Objectives 6-2
Lesson Agenda 6-3
Retrieving Data by Using a Subquery as a Source 6-4
Lesson Agenda 6-6
Multiple-Column Subqueries 6-7
Column Comparisons 6-8
Pairwise Comparison Subquery 6-9
Nonpairwise Comparison Subquery 6-10
Lesson Agenda 6-11
Scalar Subquery Expressions 6-12
Scalar Subqueries: Examples 6-13
Lesson Agenda 6-14
Correlated Subqueries 6-15
Using Correlated Subqueries: Example 1 6-17
Using Correlated Subqueries: Example 2 6-18
Lesson Agenda 6-19
Using the EXISTS Operator 6-20
Find All Departments That Do Not Have Any Employees 6-22
Lesson Agenda 6-23
WITH Clause 6-24
WITH Clause: Example 6-25
Recursive WITH Clause 6-26
Recursive WITH Clause: Example 6-27
Quiz 6-28
Summary 6-29
Practice 6: Overview 6-30

7 Manipulating Data by Using Subqueries
Objectives 7-2
Lesson Agenda 7-3
Using Subqueries to Manipulate Data 7-4
Lesson Agenda 7-5
Inserting by Using a Subquery as a Target 7-6
Lesson Agenda 7-8
Using the WITH CHECK OPTION Keyword on DML Statements 7-9
Lesson Agenda 7-11
Correlated UPDATE 7-12
Using Correlated UPDATE 7-13
Correlated DELETE 7-15
Using Correlated DELETE 7-16
Summary 7-17
Practice 7: Overview 7-18

8 Controlling User Access
Objectives 8-2
Lesson Agenda 8-3
Controlling User Access 8-4
Privileges 8-5
System Privileges 8-6
Creating Users 8-7
User System Privileges 8-8
Granting System Privileges 8-10
Lesson Agenda 8-11
What Is a Role? 8-12
Creating and Granting Privileges to a Role 8-13
Changing Your Password 8-14
Lesson Agenda 8-15
Object Privileges 8-16
Granting Object Privileges 8-18
Passing On Your Privileges 8-19
Confirming Granted Privileges 8-20
Lesson Agenda 8-21
Revoking Object Privileges 8-22
Quiz 8-24
Summary 8-25
Practice 8: Overview 8-26

9 Manipulating Data
Objectives 9-2
Lesson Agenda 9-3
Explicit Default Feature: Overview 9-4
Using Explicit Default Values 9-5
Lesson Agenda 9-6
Multitable INSERT Statements: Overview 9-7
Types of Multitable INSERT Statements 9-9
Multitable INSERT Statements 9-10
Unconditional INSERT ALL 9-12
Conditional INSERT ALL: Example 9-13
Conditional INSERT ALL 9-14
Conditional INSERT FIRST: Example 9-16
Conditional INSERT FIRST 9-17
Pivoting INSERT 9-19
Lesson Agenda 9-22
MERGE Statement 9-23
MERGE Statement Syntax 9-24
Merging Rows: Example 9-25
Lesson Agenda 9-28
FLASHBACK TABLE Statement 9-29
Using the FLASHBACK TABLE Statement 9-31
Lesson Agenda 9-32
Tracking Changes in Data 9-33
Flashback Query: Example 9-34
Flashback Version Query: Example 9-35
VERSIONS BETWEEN Clause 9-36
Quiz 9-37
Summary 9-39
Practice 9: Overview 9-40

10 Managing Data in Different Time Zones
Objectives 10-2
Lesson Agenda 10-3
Time Zones 10-4
TIME_ZONE Session Parameter 10-5
CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP 10-6
Comparing Date and Time in a Session's Time Zone 10-7
DBTIMEZONE and SESSIONTIMEZONE 10-9
TIMESTAMP Data Types 10-10
TIMESTAMP Fields 10-11

Difference Between DATE and TIMESTAMP 10-12
Comparing TIMESTAMP Data Types 10-13
Lesson Agenda 10-14
INTERVAL Data Types 10-15
INTERVAL Fields 10-17
INTERVAL YEAR TO MONTH: Example 10-18
INTERVAL DAY TO SECOND Data Type: Example 10-20
Lesson Agenda 10-21
EXTRACT 10-22
TZ_OFFSET 10-23
FROM_TZ 10-25
TO_TIMESTAMP 10-26
TO_YMINTERVAL 10-27
TO_DSINTERVAL 10-28
Daylight Saving Time (DST) 10-29
Quiz 10-31
Summary 10-32
Practice 10: Overview 10-33

A Table Descriptions

B Using SQL Developer
Objectives B-2
What Is Oracle SQL Developer? B-3
Specifications of SQL Developer B-4
SQL Developer 3.2 Interface B-5
Creating a Database Connection B-7
Browsing Database Objects B-10
Displaying the Table Structure B-11
Browsing Files B-12
Creating a Schema Object B-13
Creating a New Table: Example B-14
Using the SQL Worksheet B-15
Executing SQL Statements B-19
Saving SQL Scripts B-20
Executing Saved Script Files: Method 1 B-21
Executing Saved Script Files: Method 2 B-22
Formatting the SQL Code B-23
Using Snippets B-24
Using Snippets: Example B-25
Using Recycle Bin B-26

Debugging Procedures and Functions B-27
Database Reporting B-28
Creating a User-Defined Report B-29
Search Engines and External Tools B-30
Setting Preferences B-31
Resetting the SQL Developer Layout B-33
Data Modeler in SQL Developer B-34
Summary B-35

C Using SQL*Plus
Objectives C-2
SQL and SQL*Plus Interaction C-3
SQL Statements Versus SQL*Plus Commands C-4
Overview of SQL*Plus C-5
Logging In to SQL*Plus C-6
Displaying the Table Structure C-7
SQL*Plus Editing Commands C-9
Using LIST, n, and APPEND C-11
Using the CHANGE Command C-12
SQL*Plus File Commands C-13
Using the SAVE, START Commands C-14
SERVEROUTPUT Command C-15
Using the SQL*Plus SPOOL Command C-16
Using the AUTOTRACE Command C-17
Summary C-18

D Commonly Used SQL Commands
Objectives D-2
Basic SELECT Statement D-3
SELECT Statement D-4
WHERE Clause D-5
ORDER BY Clause D-6
GROUP BY Clause D-7
Data Definition Language D-8
CREATE TABLE Statement D-9
ALTER TABLE Statement D-10
DROP TABLE Statement D-11
GRANT Statement D-12
Privilege Types D-13
REVOKE Statement D-14
TRUNCATE TABLE Statement D-15

Data Manipulation Language D-16
INSERT Statement D-17
UPDATE Statement Syntax D-18
DELETE Statement D-19
Transaction Control Statements D-20
COMMIT Statement D-21
ROLLBACK Statement D-22
SAVEPOINT Statement D-23
Joins D-24
Types of Joins D-25
Qualifying Ambiguous Column Names D-26
Natural Join D-27
Equijoins D-28
Retrieving Records with Equijoins D-29
Additional Search Conditions Using the AND and WHERE Operators D-30
Retrieving Records with Nonequijoins D-31
Retrieving Records by Using the USING Clause D-32
Retrieving Records by Using the ON Clause D-33
Left Outer Join D-34
Right Outer Join D-35
Full Outer Join D-36
Self-Join: Example D-37
Cross Join D-38
Summary D-39

E Generating Reports by Grouping Related Data
Objectives E-2
Review of Group Functions E-3
Review of the GROUP BY Clause E-4
Review of the HAVING Clause E-5
GROUP BY with ROLLUP and CUBE Operators E-6
ROLLUP Operator E-7
ROLLUP Operator: Example E-8
CUBE Operator E-9
CUBE Operator: Example E-10
GROUPING Function E-11
GROUPING Function: Example E-12
GROUPING SETS E-13
GROUPING SETS: Example E-15
Composite Columns E-17
Composite Columns: Example E-19

Concatenated Groupings E-21
Concatenated Groupings: Example E-22
Summary E-23

F Hierarchical Retrieval
Objectives F-2
Sample Data from the EMPLOYEES Table F-3
Natural Tree Structure F-4
Hierarchical Queries F-5
Walking the Tree F-6
Walking the Tree: From the Bottom Up F-8
Walking the Tree: From the Top Down F-9
Ranking Rows with the LEVEL Pseudocolumn F-10
Formatting Hierarchical Reports Using LEVEL and LPAD F-11
Pruning Branches F-13
Summary F-14

G Writing Advanced Scripts
Objectives G-2
Using SQL to Generate SQL G-3
Creating a Basic Script G-4
Controlling the Environment G-5
The Complete Picture G-6
Dumping the Contents of a Table to a File G-7
Generating a Dynamic Predicate G-9
Summary G-11

H Oracle Database Architectural Components
Objectives H-2
Oracle Database Architecture: Overview H-3
Oracle Database Server Structures H-4
Connecting to the Database H-5
Interacting with an Oracle Database H-6
Oracle Memory Architecture H-8
Process Architecture H-10
Database Writer Process H-12
Log Writer Process H-13
Checkpoint Process H-14
System Monitor Process H-15
Process Monitor Process H-16
Oracle Database Storage Architecture H-17

Logical and Physical Database Structures H-19
Processing a SQL Statement H-21
Processing a Query H-22
Shared Pool H-23
Database Buffer Cache H-25
Program Global Area (PGA) H-26
Processing a DML Statement H-27
Redo Log Buffer H-29
Rollback Segment H-30
COMMIT Processing H-31
Summary of the Oracle Database Architecture H-33
Summary H-34

I Regular Expression Support
Objectives I-2
What Are Regular Expressions? I-3
Benefits of Using Regular Expressions I-4
Using the Regular Expressions Functions and Conditions in SQL and PL/SQL I-5
What Are Metacharacters? I-6
Using Metacharacters with Regular Expressions I-7
Regular Expressions Functions and Conditions: Syntax I-9
Performing a Basic Search by Using the REGEXP_LIKE Condition I-10
Replacing Patterns by Using the REGEXP_REPLACE Function I-11
Finding Patterns by Using the REGEXP_INSTR Function I-12
Extracting Substrings by Using the REGEXP_SUBSTR Function I-13
Subexpressions I-14
Using Subexpressions with Regular Expression Support I-15
Why Access the nth Subexpression? I-16
REGEXP_SUBSTR: Example I-17
Using the REGEXP_COUNT Function I-18
Regular Expressions and Check Constraints: Examples I-19
Quiz I-20
Summary I-21

1

Introduction

Lesson Objectives

After completing this lesson, you should be able to:

- Discuss the goals of the course
- Describe the database schema and tables that are used in the course
- Identify the available environments that can be used in the course
- Review some of the basic concepts of SQL

ORACLE

Lesson Agenda

- Course objectives and course agenda
- The database schema, the appendixes and practices, and development environments used in this course
- Review of some basic SQL concepts
- Oracle Database 12*c* documentation and additional resources

ORACLE

Course Objectives

After completing this course, you should be able to:

- Manage objects with data dictionary views
- Create schema objects
- Manage schema objects
- Write multiple-column subqueries
- Use scalar and correlated subqueries
- Control user access to specific database objects
- Add new users with different levels of access privileges
- Manipulate large data sets in the Oracle database by using subqueries
- Manage data in different time zones

Course Prerequisites

The *Oracle Database: SQL Workshop I* course is a prerequisite for this course.

The *Oracle Database: SQL Workshop I* course offers you an introduction to Oracle Database technology. In this course, you learn the basic concepts of relational databases and the powerful SQL programming language. This course provides the essential SQL skills that enable you to write queries against single and multiple tables, manipulate data in tables, create database objects, and query metadata.

Course Agenda

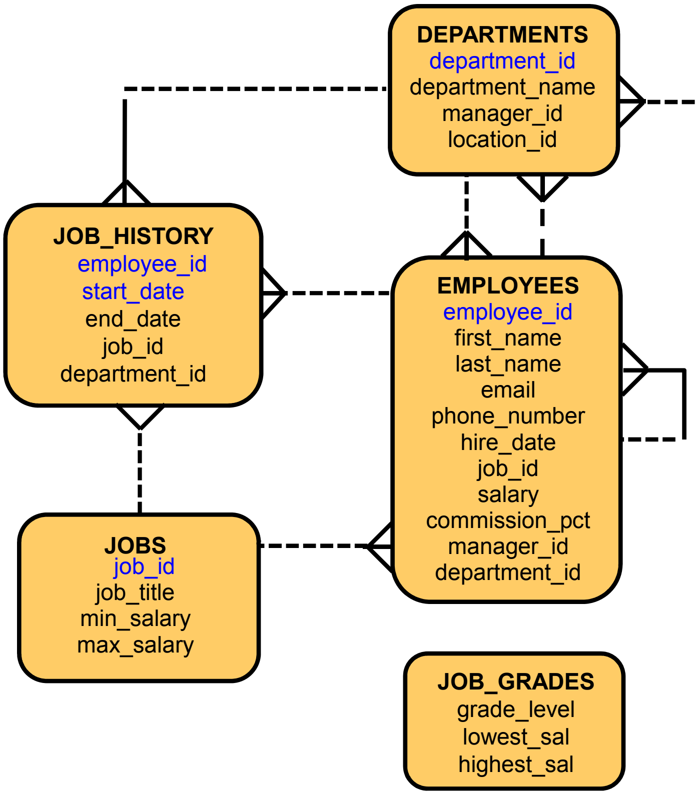
- Day 1:
  - Introduction
  - Introduction to Data Dictionary Views
  - Creating Sequence, Synonyms, and Indexes
  - Creating Views
  - Managing Schema Objects
- Day 2:
  - Retrieving Data by Using Subqueries
  - Manipulating Data by Using Subqueries
  - Controlling User Access
  - Manipulating Data
  - Managing Data in Different Time Zones

Lesson Agenda

- Course objectives and course agenda
- The database schema, the appendixes and practices, and development environments used in this course
- Review of some basic SQL concepts
- Oracle Database 12*c* documentation and additional resources

ORACLE

Tables Used in This Course



This course uses data from the following tables:

Table Descriptions

- The `EMPLOYEES` table contains information about all the employees, such as their first and last names, job `ID`s, salaries, hire dates, department `ID`s, and manager `ID`s. This table is a child of the `DEPARTMENTS` table.
- The `DEPARTMENTS` table contains information such as the department `ID`, department name, manager `ID`, and location `ID`. This table is the primary key table to the `EMPLOYEES` table.
- The `LOCATIONS` table contains department location information. It contains location `ID`, street address, city, state province, postal code, and country `ID` information. It is the primary key table to the `DEPARTMENTS` table and is a child of the `COUNTRIES` table.
- The `COUNTRIES` table contains the country names, country `ID`s, and region `ID`s. It is a child of the `REGIONS` table. This table is the primary key table to the `LOCATIONS` table.
- The `REGIONS` table contains region `ID`s and region names of the various countries. It is a primary key table to the `COUNTRIES` table.

- The `JOB_GRADES` table identifies a salary range per job grade. The salary ranges do not overlap.
- The `JOB_HISTORY` table stores job history of the employees.
- The `JOBS` table contains job titles and salary ranges.

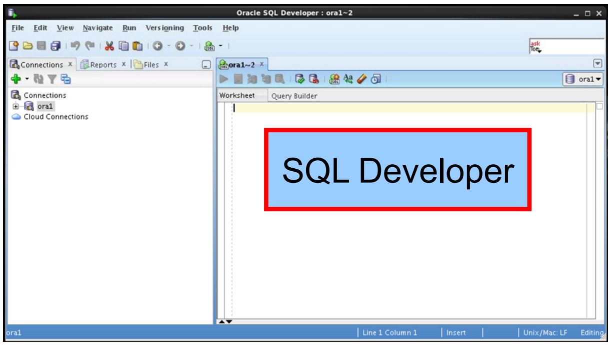
Appendixes and Practices Used in This Course

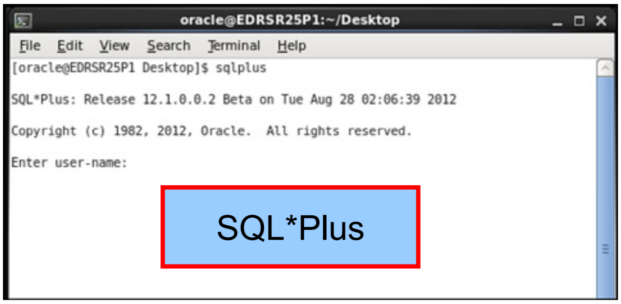
- Appendix A: Table Descriptions
- Appendix B: Using SQL Developer
- Appendix C: Using SQL* Plus
- Appendix D: Commonly Used SQL Commands
- Appendix E: Generating Reports by Grouping Related Data
- Appendix F: Hierarchical Retrieval
- Appendix G: Writing Advanced Scripts
- Appendix H: Oracle Database Architectural Components
- Appendix I : Regular Expression Support
- Practices and Solutions
- Additional Practices and Solutions

Development Environments

There are two development environments for this course:

- The primary tool is Oracle SQL Developer.
- You can also use SQL*Plus command-line interface.





SQL Developer

This course has been developed using Oracle SQL Developer as the tool for running the SQL statements discussed in the examples in the slide and the practices.

SQL*Plus

The SQL*Plus environment may also be used to run all SQL commands covered in this course.

Note

- See Appendix B titled “Using SQL Developer” for information about using SQL Developer.
- See Appendix C titled “Using SQL*Plus” for information about using SQL*Plus.

Lesson Agenda

- Course objectives and course agenda
- The database schema, the appendixes and practices, and development environments used in this course
- Review of some basic SQL concepts
- Oracle Database 12*c* documentation and additional resources

The next few slides provide a brief overview of some of the basic concepts that you learned in the course titled *Oracle Database: SQL Workshop I*.

Review of Restricting Data

- Restrict the rows that are returned by using the `WHERE` clause.
- Use comparison conditions to compare one expression with another value or expression.

Operator	Meaning
`BETWEEN ...AND...`	Between two values (inclusive)
`IN(set)`	Match any of a list of values
`LIKE`	Match a character pattern

- Use logical conditions to combine the result of two component conditions and produce a single result based on those conditions.

You can restrict the rows that are returned from the query by using the `WHERE` clause. A `WHERE` clause contains a condition that must be met, and it directly follows the `FROM` clause.

The `WHERE` clause can compare values in columns, literal values, arithmetic expression, or functions. It consists of three elements:

- Column name
- Comparison condition
- Column name, constant, or list of values

You use comparison conditions in the `WHERE` clause in the following format:

```
... WHERE expr operator value
```

Apart from those mentioned in the slide, you use other comparison conditions such as `=`, `<`, `>`, `<>`, `<=`, and `>=`.

Three logical operators are available in SQL:

- `AND`
- `OR`
- `NOT`

Review of Sorting Data

- Sort retrieved rows with the `ORDER BY` clause:
  - `ASC`: Ascending order, default
  - `DESC`: Descending order
- The `ORDER BY` clause comes last in the `SELECT` statement:

```
SELECT   last_name, job_id, department_id, hire_date
FROM     employees
ORDER BY hire_date ;
```

	LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
1	De Haan	AD_VP	90	13-JAN-01
2	Gietz	AC_ACCOUNT	110	07-JUN-02
3	Baer	PR_REP	70	07-JUN-02
4	Mavris	HR_REP	40	07-JUN-02
5	Higgins	AC_MGR	110	07-JUN-02
6	Faviet	FI_ACCOUNT	100	16-AUG-02
7	Greenberg	FI_MGR	100	17-AUG-02

...

The order of rows that are returned in a query result is undefined. The `ORDER BY` clause can be used to sort the rows. If you use the `ORDER BY` clause, it must be the last clause of the SQL statement. You can specify an expression, an alias, or a column position as the sort condition.

Syntax

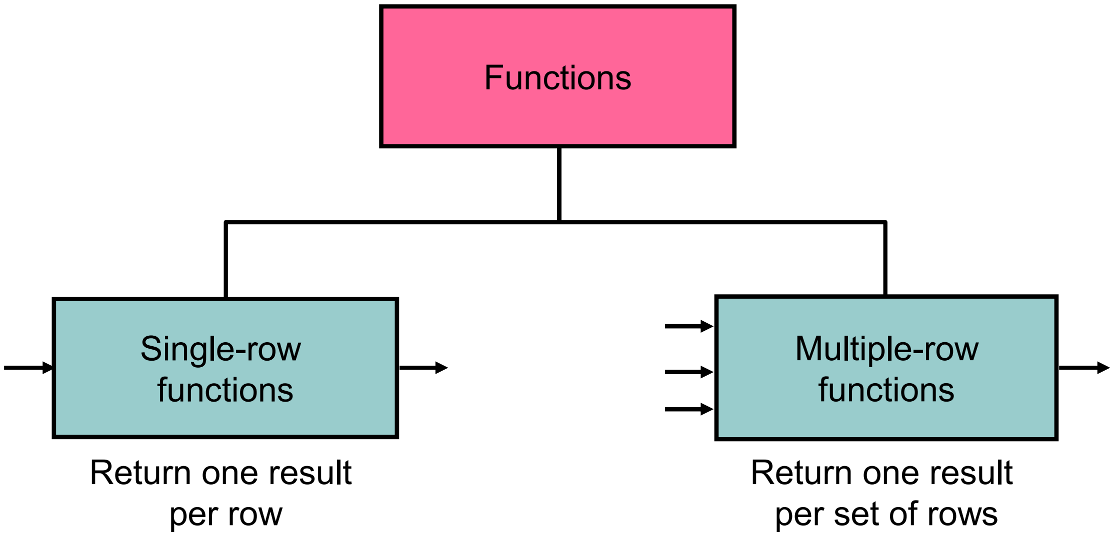
```
SELECT          expr
FROM            table
[WHERE          condition(s)]
[ORDER BY    {column, expr, numeric_position} [ASC|DESC]];
```

In the syntax:

`ORDER BY`	Specifies the order in which the retrieved rows are displayed
`ASC`	Orders the rows in ascending order (This is the default order.)
`DESC`	Orders the rows in descending order

If the `ORDER BY` clause is not used, the sort order is undefined, and the Oracle Server may not fetch rows in the same order for the same query twice. Use the `ORDER BY` clause to display the rows in a specific order.

Review of SQL Functions



There are two types of functions:

- Single-row functions
- Multiple-row functions

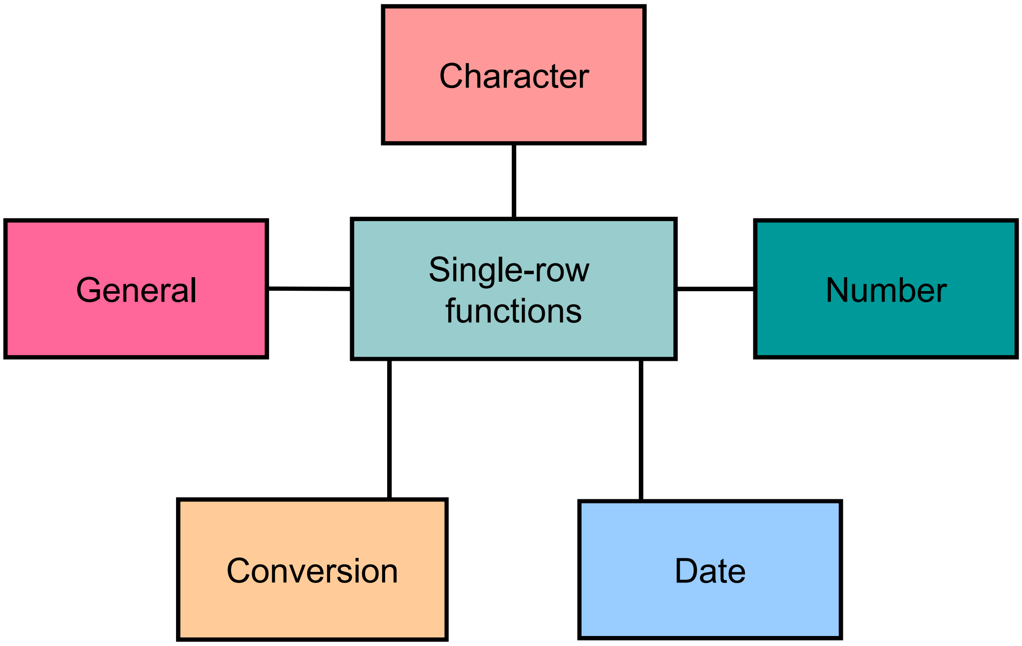
Single-Row Functions

These functions operate only on single rows and return one result per row. There are different types of single-row functions such as character, number, date, conversion, and general functions.

Multiple-Row Functions

Functions can manipulate groups of rows to give one result per group of rows. These functions are also known as *group functions*.

Review of Single-Row Functions

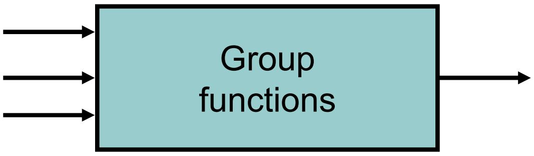


The following are different types of single-row functions:

- **Character functions:** Accept character input and can return both character and number values
- **Number functions:** Accept numeric input and return numeric values
- **Date functions:** Operate on values of the `DATE` data type (All date functions return a value of the `DATE` data type, except the `MONTHS_BETWEEN` function, which returns a number.)
- **Conversion functions:** Convert a value from one data type to another
- **General functions:**
  - `NVL`
  - `NVL2`
  - `NULLIF`
  - `COALESCE`
  - `CASE`
  - `DECODE`

Review of Types of Group Functions

- AVG
- COUNT
- MAX
- MIN
- STDDEV
- SUM
- VARIANCE



Each of the functions accepts an argument. The following table identifies the options that you can use in the syntax:

<table>
<tr><th>Function</th><th>Description</th></tr>
<tr><td>AVG([DISTINCT|ALL]n)</td><td>Average value of n, ignoring null values</td></tr>
<tr><td>COUNT({*|[DISTINCT|ALL]expr})</td><td>Number of rows, where expr evaluates to something other than null (count all selected rows using *, including duplicates and rows with nulls)</td></tr>
<tr><td>MAX([DISTINCT|ALL]expr)</td><td>Maximum value of expr, ignoring null values</td></tr>
<tr><td>MIN([DISTINCT|ALL]expr)</td><td>Minimum value of expr, ignoring null values</td></tr>
<tr><td>STDDEV([DISTINCT|ALL]n)</td><td>Standard deviation of n, ignoring null values</td></tr>
<tr><td>SUM([DISTINCT|ALL]n)</td><td>Sum values of n, ignoring null values</td></tr>
<tr><td>VARIANCE([DISTINCT|ALL]n)</td><td>Variance of n, ignoring null values</td></tr>
</table>

Review of Using Subqueries

- A subquery is a `SELECT` statement nested in a clause of another `SELECT` statement.
- Syntax:

```
SELECT select_list
FROM   table
WHERE  expr operator
                          (SELECT select_list
                           FROM   table   );
```

- Types of subqueries:

Single-row subquery	Multiple-row subquery
Returns only one row	Returns more than one row
Uses single-row comparison operators	Uses multiple-row comparison operators

You can build powerful statements out of simple ones by using subqueries. Subqueries are useful when a query is based on a search criterion with unknown intermediate values.

You can place the subquery in a number of SQL clauses, including the following:

- `WHERE` clause
- `HAVING` clause
- `FROM` clause

The subquery (inner query) executes once before the main query (outer query). The result of the subquery is used by the main query.

A single-row subquery uses a single-row operator such as `=`, `>`, `<`, `>=`, `<=`, or `<>`. With a multiple-row subquery, you use a multiple-row operator such as `IN`, `ANY`, or `ALL`.

Example: Display details of employees whose salary is equal to the minimum salary.

```
SELECT last_name, salary, job_id
FROM   employees
WHERE  salary = (SELECT MIN(salary)
                 FROM   employees );
```

In the example, the `MIN` group function returns a single value to the outer query.

Note: In this course, you learn how to use multiple-column subqueries. Multiple-column subqueries return more than one column from the inner `SELECT` statement.

Review of Managing Tables Using DML Statements

A data manipulation language (DML) statement is executed when you:

- Add new rows to a table
- Modify existing rows in a table
- Remove existing rows from a table

Function	Description
`INSERT`	Adds a new row to the table
`UPDATE`	Modifies existing rows in the table
`DELETE`	Removes existing rows from the table
`MERGE`	Updates, inserts, or deletes a row conditionally into/from a table

When you want to add, update, or delete data in the database, you execute a DML statement. A collection of DML statements that form a logical unit of work is called a transaction. You can add new rows to a table by using the `INSERT` statement. With the following syntax, only one row is inserted at a time.

```
INSERT INTO table [(column [, column…])]
VALUES      (value[, value...]);
```

You can use the `INSERT` statement to add rows to a table where the values are derived from existing tables. In place of the `VALUES` clause, you use a subquery. The number of columns and their data types in the column list of the `INSERT` clause must match the number of values and their data types in the subquery.

The `UPDATE` statement modifies specific rows if you specify the `WHERE` clause.

```
UPDATE table
SET column = value [, column = value, ...]
[WHERE condition];
```

You can remove existing rows by using the DELETE statement. You can delete specific rows by specifying the WHERE clause in the DELETE statement.

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

You learn about the MERGE statement in the lesson titled “Manipulating Data.”

Lesson Agenda

- Course objectives and course agenda
- The database schema, the appendixes and practices, and development environments used in this course
- Review of some basic SQL concepts
- Oracle Database 12*c* documentation and additional resources

Oracle Database SQL Documentation

- *Oracle Database New Features Guide*
- *Oracle Database Reference*
- *Oracle Database SQL Language Reference*
- *Oracle Database Concepts*
- *Oracle Database SQL Developer User's Guide Release 3.2*

Navigate to http://www.oracle.com/pls/db121/homepage to access the Oracle Database 12*c* Release 1 documentation library.

Additional Resources

For additional information about the new Oracle 12*c* SQL, refer to the following:

- *Oracle Database 12c: New Features Self Studies*
- *Oracle by Example series (OBE): Oracle Database 12c*
- *Oracle Learning Library:*
  – http://www.oracle.com/goto/oll
- Access the online SQL Developer Home Page, which is available at:
  – http://www.oracle.com/technology/products/database/sql_developer/index.html
- Access the SQL Developer tutorial, which is available online at
  – http://download.oracle.com/oll/tutorials/SQLDeveloper/index.htm

Summary

In this lesson, you should have learned how to:

- Discuss the goals of the course
- Describe the database schema and tables that are used in the course
- Identify the available environments that can be used in the course
- Recall some of the basic concepts of SQL

Practice 1: Overview

This practice covers the following topics:

- Running the SQL Developer online tutorial
- Starting SQL Developer and creating a new database connection and browsing the tables
- Executing SQL statements using the SQL Worksheet
- Running some basic SQL commands

In this practice, you use SQL Developer to execute SQL statements.

Note: All written practices use SQL Developer as the development environment. Although it is recommended that you use SQL Developer, you can also use the SQL*Plus environment that is available in this course.

Introduction to Data Dictionary Views

Objectives

After completing this lesson, you should be able to:

- Use the data dictionary views to research data on your objects
- Query various data dictionary views

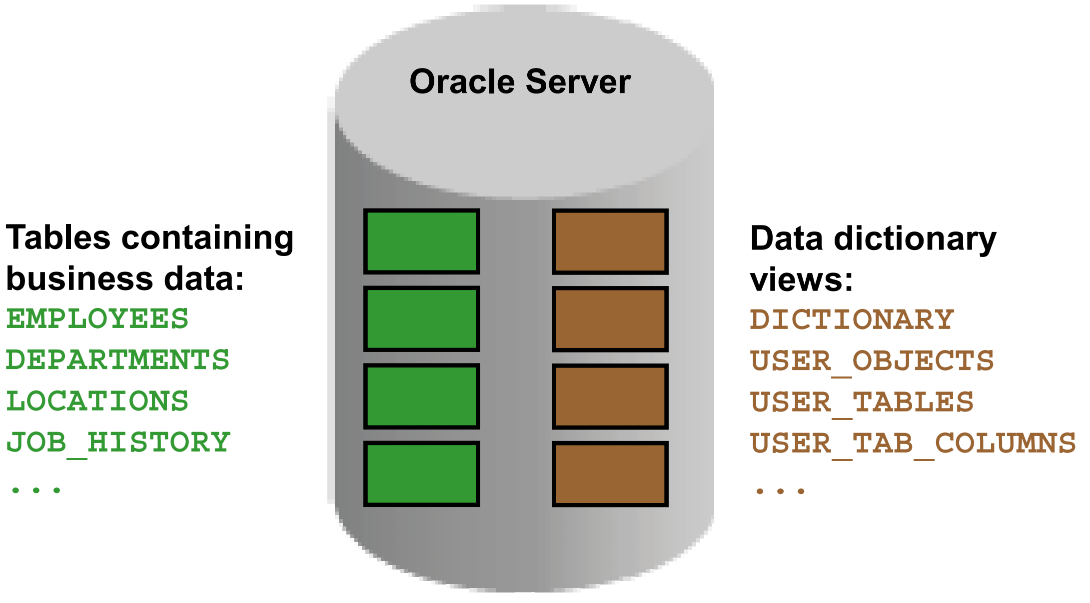
ORACLE

In this lesson, you are introduced to the data dictionary views. You learn that the dictionary views can be used to retrieve metadata and create reports about your schema objects.

Lesson Agenda

- Introduction to data dictionary
- Querying the dictionary views for the following:
  - Table information
  - Column information
  - Constraint information
- Adding a comment to a table and querying the dictionary views for comment information

Data Dictionary



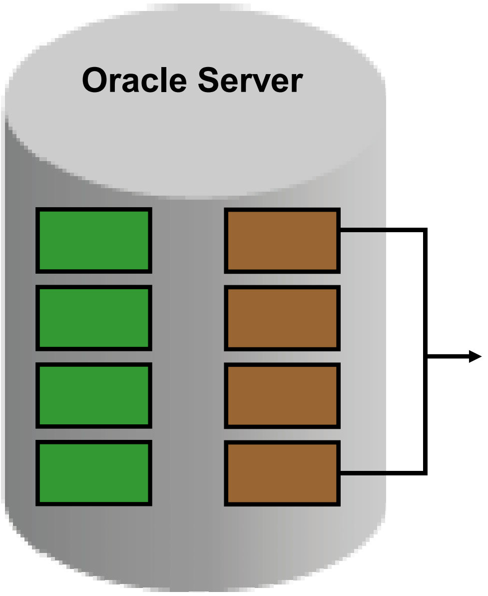
User tables are tables created by the user and contain business data, such as `EMPLOYEES`. There is another collection of tables and views in the Oracle database known as the *data dictionary*. This collection is created and maintained by the Oracle Server and contains information about the database. The data dictionary is structured in tables and views, just like other database data. Not only is the data dictionary central to every Oracle database, but it is also an important tool for all users, from end users to application designers and database administrators.

You use SQL statements to access the data dictionary. Because the data dictionary is read-only, you can issue only queries against its tables and views.

You can query the dictionary views that are based on the dictionary tables to find information such as:

- Definitions of all schema objects in the database (tables, views, indexes, synonyms, sequences, procedures, functions, packages, triggers, and so on)
- Default values for columns
- Integrity constraint information
- Names of Oracle users
- Privileges and roles that each user has been granted
- Other general database information

Data Dictionary Structure



Consists of:

- **Base tables**
- **User-accessible views**

Underlying base tables store information about the associated database. Only the Oracle Server should write to and read from these tables. You rarely access them directly.

There are several views that summarize and display the information stored in the base tables of the data dictionary. These views decode the base table data into useful information (such as user or table names) using joins and `WHERE` clauses to simplify the information. Most users are given access to the views rather than the base tables.

The Oracle user `SYS` owns all base tables and user-accessible views of the data dictionary. No Oracle user should *ever* alter (`UPDATE`, `DELETE`, or `INSERT`) any rows or schema objects contained in the `SYS` schema, because such activity can compromise data integrity.

Data Dictionary Structure

View naming convention:

View Prefix	Purpose
USER	User's view (what is in your schema; what you own)
ALL	Expanded user's view (what you can access)
DBA	Database administrator's view (what is in everyone's schemas)
V$	Performance-related data

The data dictionary consists of sets of views. In many cases, a set consists of three views containing similar information and distinguished from each other by their prefixes. For example, there is a view named `USER_OBJECTS`, another named `ALL_OBJECTS`, and a third named `DBA_OBJECTS`.

These three views contain similar information about objects in the database, except that the scope is different. `USER_OBJECTS` contains information about objects that you own or you created. `ALL_OBJECTS` contains information about all objects to which you have access. `DBA_OBJECTS` contains information about all objects that are owned by all users. For views that are prefixed with `ALL` or `DBA`, there is usually an additional column in the view named `OWNER` to identify who owns the object.

There is also a set of views that is prefixed with `v$`. These views are dynamic in nature and hold information about performance. Dynamic performance tables are not true tables, and they should not be accessed by most users. However, database administrators can query and create views on the tables and grant access to those views to other users. This course does not go into details about these views.

How to Use the Dictionary Views

Start with `DICTIONARY`. It contains the names and descriptions of the dictionary tables and views.

```
DESCRIBE DICTIONARY
```

```
DESCRIBE dictionary
Name       Null Type
---------- ---- --------------
TABLE_NAME      VARCHAR2(128)
COMMENTS        VARCHAR2(4000)
```

```
SELECT *
FROM   dictionary
WHERE  table_name = 'USER_OBJECTS';
```

	TABLE_NAME	COMMENTS
1	USER_OBJECTS	Objects owned by the user

To familiarize yourself with the dictionary views, you can use the dictionary view named `DICTIONARY`. It contains the name and short description of each dictionary view to which you have access.

You can write queries to search for information about a particular view name, or you can search the `COMMENTS` column for a word or phrase. In the example shown, the `DICTIONARY` view is described. It has two columns. The `SELECT` statement retrieves information about the dictionary view named `USER_OBJECTS`. The `USER_OBJECTS` view contains information about all the objects that you own.

You can write queries to search the `COMMENTS` column for a word or phrase. For example, the following query returns the names of all views that you are permitted to access in which the `COMMENTS` column contains the word *columns*:

```
SELECT table_name
FROM dictionary
WHERE LOWER(comments) LIKE '%columns%';
```

Note: The names in the data dictionary are in uppercase.

USER_OBJECTS and ALL_OBJECTS Views

USER_OBJECTS:

- Query USER_OBJECTS to see all the objects that you own.
- Using USER_OBJECTS, you can obtain a listing of all object names and types in your schema, plus the following information:
  - Date created
  - Date of last modification
  - Status (valid or invalid)

ALL_OBJECTS:

- Query ALL_OBJECTS to see all the objects to which you have access.

You can query the USER_OBJECTS view to see the names and types of all the objects in your schema. There are several columns in this view:

- **OBJECT_NAME:** Name of the object
- **OBJECT_ID:** Dictionary object number of the object
- **OBJECT_TYPE:** Type of object (such as TABLE, VIEW, INDEX, SEQUENCE)
- **CREATED:** Time stamp for the creation of the object
- **LAST_DDL_TIME:** Time stamp for the last modification of the object resulting from a data definition language (DDL) command
- **STATUS:** Status of the object (VALID, INVALID, or N/A)
- **GENERATED:** Was the name of this object system-generated? (Y|N)

Note: This is not a complete listing of the columns. For a complete listing, see "USER_OBJECTS" in the *Oracle® Database Reference 12c Release 1***.**

You can also query the ALL_OBJECTS view to see a listing of all objects to which you have access.

USER_OBJECTS View

```
SELECT object_name, object_type, created, status
FROM   user_objects
ORDER BY object_type;
```

	OBJECT_NAME	OBJECT_TYPE	CREATED	STATUS
1	JHIST_EMPLOYEE_IX	INDEX	23-AUG-12	VALID
2	EMP_DEPARTMENT_IX	INDEX	23-AUG-12	VALID
3	LOC_CITY_IX	INDEX	23-AUG-12	VALID
4	LOC_STATE_PROVINCE_IX	INDEX	23-AUG-12	VALID
5	LOC_COUNTRY_IX	INDEX	23-AUG-12	VALID
6	JHIST_DEPARTMENT_IX	INDEX	23-AUG-12	VALID
7	COUNTRY_C_ID_PK	INDEX	23-AUG-12	VALID
8	JHIST_EMP_ID_ST_DATE_PK	INDEX	23-AUG-12	VALID

…

The example shows the names, types, dates of creation, and status of all objects that are owned by this user.

The OBJECT_TYPE column holds the values of either TABLE, VIEW, SEQUENCE, INDEX, PROCEDURE, FUNCTION, PACKAGE, or TRIGGER.

The STATUS column holds a value of VALID, INVALID, or N/A. Although tables are always valid, the views, procedures, functions, packages, and triggers may be invalid.

The CAT View

For a simplified query and output, you can query the CAT view. This view contains only two columns: TABLE_NAME and TABLE_TYPE. It provides the names of all your INDEX, TABLE, CLUSTER, VIEW, SYNONYM, SEQUENCE, or UNDEFINED objects.

Note: CAT is a synonym for USER_CATALOG—a view that lists tables, views, synonyms and sequences owned by the user.

Lesson Agenda

- Introduction to data dictionary
- Querying the dictionary views for the following:
  - Table information
  - Column information
  - Constraint information
- Adding a comment to a table and querying the dictionary views for comment information

Table Information

USER_TABLES:

```
DESCRIBE user_tables
```

```
DESCRIBE user_tables
Name                     Null     Type
------------------------ -------- -------------
TABLE_NAME               NOT NULL VARCHAR2(128)
TABLESPACE_NAME                   VARCHAR2(30)
CLUSTER_NAME                      VARCHAR2(128)
IOT_NAME                          VARCHAR2(128)
```

...

```
SELECT table_name
FROM   user_tables;
```

	TABLE_NAME
1	REGIONS
2	LOCATIONS
3	DEPARTMENTS
4	JOBS
5	EMPLOYEES
6	JOB_HISTORY

...

You can use the USER_TABLES view to obtain the names of all your tables. The USER_TABLES view contains information about your tables. In addition to providing the table name, it contains detailed information about the storage.

The TABS view is a synonym of the USER_TABLES view. You can query it to see a listing of tables that you own:

```
SELECT table_name
FROM   tabs;
```

Note: For a complete listing of the columns in the USER_TABLES view, see "USER_TABLES" in the *Oracle® Database Reference 12c Release 1.*

You can also query the ALL_TABLES view to see a listing of all tables to which you have access.

Column Information

USER_TAB_COLUMNS:

```
DESCRIBE user_tab_columns
```

```
Name                 Null     Type
-------------------- -------- -------------
TABLE_NAME           NOT NULL VARCHAR2(128)
COLUMN_NAME          NOT NULL VARCHAR2(128)
DATA_TYPE                     VARCHAR2(128)
DATA_TYPE_MOD                 VARCHAR2(3)
DATA_TYPE_OWNER               VARCHAR2(128)
DATA_LENGTH          NOT NULL NUMBER
DATA_PRECISION                NUMBER
DATA_SCALE                    NUMBER
NULLABLE                      VARCHAR2(1)
```

...

You can query the USER_TAB_COLUMNS view to find detailed information about the columns in your tables. Although the USER_TABLES view provides information about your table names and storage, detailed column information is found in the USER_TAB_COLUMNS view.

This view contains information such as:

- Column names
- Column data types
- Length of data types
- Precision and scale for NUMBER columns
- Whether nulls are allowed (Is there a NOT NULL constraint on the column?)
- Default value

Note: For a complete listing and description of the columns in the USER_TAB_COLUMNS view, see "USER_TAB_COLUMNS" in the *Oracle® Database Reference 12c Release 1.*

Column Information

```
SELECT column_name, data_type, data_length,
       data_precision, data_scale, nullable
FROM   user_tab_columns
WHERE  table_name = 'EMPLOYEES';
```

	COLUMN_NAME	DATA_TYPE	DATA_LENGTH	DATA_PRECISION	DATA_SCALE	NULLABLE
1	EMPLOYEE_ID	NUMBER	22	6	0	N
2	FIRST_NAME	VARCHAR2	20	(null)	(null)	Y
3	LAST_NAME	VARCHAR2	25	(null)	(null)	N
4	EMAIL	VARCHAR2	25	(null)	(null)	N
5	PHONE_NUMBER	VARCHAR2	20	(null)	(null)	Y
6	HIRE_DATE	DATE	7	(null)	(null)	N
7	JOB_ID	VARCHAR2	10	(null)	(null)	N
8	SALARY	NUMBER	22	8	2	Y
9	COMMISSION_PCT	NUMBER	22	2	2	Y
10	MANAGER_ID	NUMBER	22	6	0	Y
11	DEPARTMENT_ID	NUMBER	22	4	0	Y

By querying the USER_TAB_COLUMNS table, you can find details about your columns such as the names, data types, data type lengths, null constraints, and default value for a column.

The example shown displays the columns, data types, data lengths, and null constraints for the EMPLOYEES table. Note that this information is similar to the output from the DESCRIBE command.

To view information about columns set as unused, you use the USER_UNUSED_COL_TABS dictionary view.

Note: Names of the objects in Data Dictionary are in uppercase.

Constraint Information

- `USER_CONSTRAINTS` describes the constraint definitions on your tables.
- `USER_CONS_COLUMNS` describes columns that are owned by you and that are specified in constraints.

```
DESCRIBE user_constraints
```

```
DESCRIBE user_constraints
Name              Null     Type
----------------- -------- -------------
OWNER                      VARCHAR2(128)
CONSTRAINT_NAME   NOT NULL VARCHAR2(128)
CONSTRAINT_TYPE            VARCHAR2(1)
TABLE_NAME        NOT NULL VARCHAR2(128)
SEARCH_CONDITION           LONG()
R_OWNER                    VARCHAR2(128)
R_CONSTRAINT_NAME          VARCHAR2(128)
DELETE_RULE                VARCHAR2(9)
STATUS                     VARCHAR2(8)
```

…

You can find out the names of your constraints, the type of constraint, the table name to which the constraint applies, the condition for check constraints, foreign key constraint information, deletion rule for foreign key constraints, the status, and many other types of information about your constraints.

Note: For a complete listing and description of the columns in the `USER_CONSTRAINTS` view, see "`USER_CONSTRAINTS`" in the *Oracle® Database Reference 12c Release 1.*

USER_CONSTRAINTS: Example

```
SELECT constraint_name, constraint_type,
       search_condition, r_constraint_name,
       delete_rule, status
FROM   user_constraints
WHERE  table_name = 'EMPLOYEES';
```

	CONSTRAINT_NAME	CONSTRAINT_TYPE	SEARCH_CONDITION	R_CONSTRAINT_NAME	DELETE_RULE	STATUS
1	EMP_MANAGER_FK	R	(null)	EMP_EMP_ID_PK	NO ACTION	ENABLED
2	EMP_JOB_FK	R	(null)	JOB_ID_PK	NO ACTION	ENABLED
3	EMP_DEPT_FK	R	(null)	DEPT_ID_PK	NO ACTION	ENABLED
4	EMP_EMP_ID_PK	P	(null)	(null)	(null)	ENABLED
5	EMP_EMAIL_UK	U	(null)	(null)	(null)	ENABLED
6	EMP_SALARY_MIN	C	salary > 0	(null)	(null)	ENABLED
7	EMP_JOB_NN	C	"JOB_ID" IS NOT NULL	(null)	(null)	ENABLED
8	EMP_HIRE_DATE_NN	C	"HIRE_DATE" IS NOT NULL	(null)	(null)	ENABLED
9	EMP_EMAIL_NN	C	"EMAIL" IS NOT NULL	(null)	(null)	ENABLED
10	EMP_LAST_NAME_NN	C	"LAST_NAME" IS NOT NULL	(null)	(null)	ENABLED

In the example shown, the USER_CONSTRAINTS view is queried to find the names, types, check conditions, name of the unique constraint that the foreign key references, deletion rule for a foreign key, and status for constraints on the EMPLOYEES table.

The CONSTRAINT_TYPE can be:

- C (check constraint on a table, or NOT NULL)
- P (primary key)
- U (unique key)
- R (referential integrity)
- V (with check option, on a view)
- O (with read-only, on a view)

The DELETE_RULE can be:

- **CASCADE:** If the parent record is deleted, the child records are deleted, too.
- **SET NULL:** If the parent record is deleted, change the respective child record to null.
- **NO ACTION:** A parent record can be deleted only if no child records exist.

The STATUS can be:

- **ENABLED:** Constraint is active.
- **DISABLED:** Constraint is made not active.

Querying USER_CONS_COLUMNS

```
DESCRIBE user_cons_columns
```

```
DESCRIBE user_cons_columns
Name            Null     Type
--------------- -------- --------------
OWNER           NOT NULL VARCHAR2(128)
CONSTRAINT_NAME NOT NULL VARCHAR2(128)
TABLE_NAME      NOT NULL VARCHAR2(128)
COLUMN_NAME              VARCHAR2(4000)
POSITION                 NUMBER
```

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

	CONSTRAINT_NAME	COLUMN_NAME
1	EMP_LAST_NAME_NN	LAST_NAME
2	EMP_EMAIL_NN	EMAIL
3	EMP_HIRE_DATE_NN	HIRE_DATE
4	EMP_JOB_NN	JOB_ID
5	EMP_SALARY_MIN	SALARY
6	EMP_EMAIL_UK	EMAIL
7	EMP_EMP_ID_PK	EMPLOYEE_ID
8	EMP_DEPT_FK	DEPARTMENT_ID
9	EMP_JOB_FK	JOB_ID
10	EMP_MANAGER_FK	MANAGER_ID

To find the names of the columns to which a constraint applies, query the USER_CONS_COLUMNS dictionary view. This view tells you the name of the owner of a constraint, the name of the constraint, the table that the constraint is on, the names of the columns with the constraint, and the original position of column or attribute in the definition of the object.

Note: A constraint may apply to more than one column.

You can also write a join between USER_CONSTRAINTS and USER_CONS_COLUMNS to create customized output from both tables.

Lesson Agenda

- Introduction to data dictionary
- Querying the dictionary views for the following:
  - Table information
  - Column information
  - Constraint information
- Adding a comment to a table and querying the dictionary views for comment information

Adding Comments to a Table

- You can add comments to a table or column by using the `COMMENT` statement:

```
COMMENT ON TABLE employees
IS 'Employee Information';
```

```
COMMENT ON COLUMN employees.first_name
IS 'First name of the employee';
```

- Comments can be viewed through the data dictionary views:
  - `ALL_COL_COMMENTS`
  - `USER_COL_COMMENTS`
  - `ALL_TAB_COMMENTS`
  - `USER_TAB_COMMENTS`

You can add a comment of up to 4,000 bytes about a column, table, view, or snapshot by using the `COMMENT` statement. The comment is stored in the data dictionary and can be viewed in one of the following data dictionary views in the `COMMENTS` column:

- `ALL_COL_COMMENTS`
- `USER_COL_COMMENTS`
- `ALL_TAB_COMMENTS`
- `USER_TAB_COMMENTS`

Syntax

```
COMMENT ON {TABLE table | COLUMN table.column}
    IS 'text';
```

In the syntax:

table	Is the name of the table
column	Is the name of the column in a table
text	Is the text of the comment

You can drop a comment from the database by setting it to empty string (`''`):

```
COMMENT ON TABLE  employees IS '';
```

Quiz

The dictionary views that are based on the dictionary tables contain information such as:

a. Definitions of all the schema objects in the database
b. Default values for the columns
c. Integrity constraint information
d. Privileges and roles that each user has been granted
e. All of the above

Answer: e

Summary

In this lesson, you should have learned how to find information about your objects through the following dictionary views:

- `DICTIONARY`
- `USER_OBJECTS`
- `USER_TABLES`
- `USER_TAB_COLUMNS`
- `USER_CONSTRAINTS`
- `USER_CONS_COLUMNS`

In this lesson, you learned about some of the dictionary views that are available to you. You can use these dictionary views to find information about your tables, constraints, views, sequences, and synonyms.

Practice 2: Overview

This practice covers the following topics:

- Querying the dictionary views for table and column information
- Querying the dictionary views for constraint information
- Adding a comment to a table and querying the dictionary views for comment information

In this practice, you query the dictionary views to find information about objects in your schema.

3

Creating Sequences, Synonyms, and Indexes

Objectives

After completing this lesson, you should be able to:

- Create, maintain, and use sequences
- Create private and public synonyms
- Create and maintain indexes
- Query various data dictionary views to find information for sequences, synonyms, and indexes

In this lesson, you are introduced to the sequence, synonyms, and index objects. You learn the basics of creating and using sequences, synonyms and indexes.

Lesson Agenda

- Overview of sequences:
  - Creating, using, and modifying a sequence
  - Cache sequence values
  - `NEXTVAL` and `CURRVAL` pseudocolumns
  - SQL column defaulting using a sequence
- Overview of synonyms
  - Creating, dropping synonyms
- Overview of indexes
  - Creating indexes
  - Using the `CREATE TABLE` statement
  - Creating function-based indexes
  - Creating multiple indexes on the same set of columns
  - Removing indexes

ORACLE

Database Objects

Object	Description
Table	Basic unit of storage; composed of rows
View	Logically represents subsets of data from one or more tables
Sequence	Generates numeric values
Index	Improves the performance of data retrieval queries
Synonym	Gives alternative names to objects

There are several other objects in a database in addition to tables.

With views, you can present and hide data from the tables.

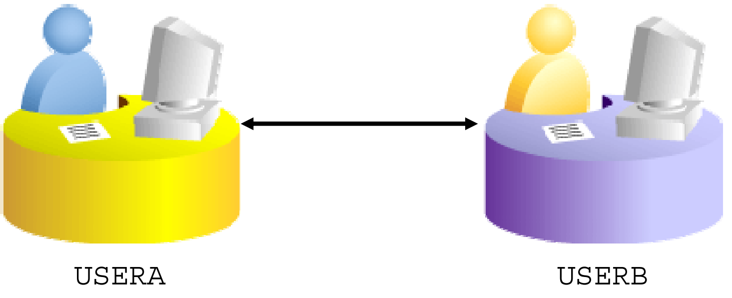
Many applications require the use of unique numbers as primary key values. You can either build code into the application to handle this requirement or use a sequence to generate unique numbers.

If you want to improve the performance of data retrieval queries, you should consider creating an index. You can also use indexes to enforce uniqueness on a column or a collection of columns.

You can provide alternative names for objects by using synonyms.

Referencing Another User's Tables

- Tables belonging to other users are not in the user's schema.
- You should use the owner's name as a prefix to those tables.




```
SELECT *
FROM userB.employees;
```

```
SELECT *
FROM userA.employees;
```


A schema is a collection of logical structures of data or *schema objects*. A schema is owned by a database user and has the same name as that user. Each user owns a single schema.

Schema objects can be created and manipulated with SQL and include tables, views, synonyms, sequences, stored procedures, indexes, clusters, and database links.

If a table does not belong to the user, the owner's name must be prefixed to the table. For example, if there are schemas named `USERA` and `USERB`, and both have an `EMPLOYEES` table, then if `USERA` wants to access the `EMPLOYEES` table that belongs to `USERB`, `USERA` must prefix the table name with the schema name:

```
SELECT *
FROM   userb.employees;
```

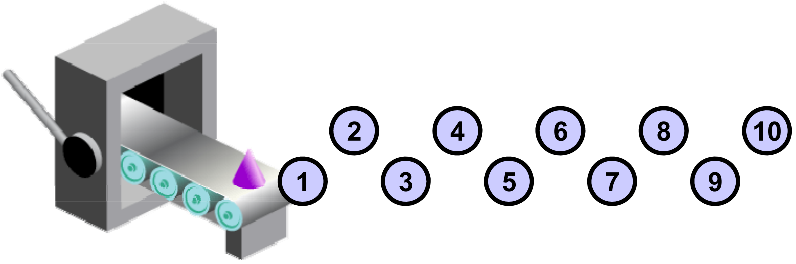
If `USERB` wants to access the `EMPLOYEES` table that is owned by `USERA`, `USERB` must prefix the table name with the schema name:

```
SELECT *
FROM   usera.employees;
```

Sequences

A sequence:

- Can automatically generate unique numbers
- Is a shareable object
- Can be used to create a primary key value
- Replaces application code
- Speeds up the efficiency of accessing sequence values when cached in memory



A sequence is a user-created database object that can be shared by multiple users to generate integers.

You can define a sequence to generate unique values or to recycle and use the same numbers again.

A typical usage for sequences is to create a primary key value, which must be unique for each row. A sequence is generated and incremented (or decremented) by an internal Oracle routine. This can be a time-saving object, because it can reduce the amount of application code needed to write a sequence-generating routine.

Sequence numbers are stored and generated independent of tables. Therefore, the same sequence can be used for multiple tables.

`CREATE SEQUENCE` Statement: Syntax

Define a sequence to generate sequential numbers automatically:

```
CREATE SEQUENCE [ schema. ] sequence
   [ { START WITH|INCREMENT BY } integer
   | { MAXVALUE integer | NOMAXVALUE }
   | { MINVALUE integer | NOMINVALUE }
   | { CYCLE | NOCYCLE }
   | { CACHE integer | NOCACHE }
   | { ORDER | NOORDER }
];
```


Automatically generate sequential numbers by using the `CREATE SEQUENCE` statement.

In the syntax:

Sequence	Is the name of the sequence generator
START WITH *n*	Specifies the first sequence number to be generated (If this clause is omitted, the sequence starts with 1.)
INCREMENT BY *n*	Specifies the interval between sequence numbers, where *n* is an integer (If this clause is omitted, the sequence increments by 1.)
MAXVALUE *n*	Specifies the maximum value the sequence can generate
NOMAXVALUE	Specifies 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 sequence value
NOMINVALUE	Specifies a minimum value of 1 for an ascending sequence and –(10^26) for a descending sequence (This is the default option.)

`ORDER`	Specify `ORDER` to guarantee that sequence numbers are generated in order of request. This clause is useful if you are using the sequence numbers as timestamps.
`NOORDER`	Specify `NOORDER` if you do not want to guarantee that sequence numbers are generated in order of request. This is the default.
`CYCLE | NOCYCLE`	Specifies whether the sequence continues to generate values after reaching its maximum or minimum value (`NOCYCLE` is the default option.)
`CACHE` *n* `| NOCACHE`	Specifies how many values the Oracle Server pre-allocates and keeps in memory (By default, the Oracle server caches 20 values.)

Creating a Sequence

- Create a sequence named `DEPT_DEPTID_SEQ` to be used for the primary key of the `DEPARTMENTS` table.
- Do not use the `CYCLE` option.

```
CREATE SEQUENCE dept_deptid_seq
                START WITH 280
                INCREMENT BY 10
                MAXVALUE 9999
                NOCACHE
                NOCYCLE;
```

```
sequence DEPT_DEPTID_SEQ created.
```


The example in the slide creates a sequence named `DEPT_DEPTID_SEQ` to be used for the `DEPARTMENT_ID` column of the `DEPARTMENTS` table. The sequence starts at 280, does not allow caching, and does not cycle.

Do not use the `CYCLE` option if the sequence is used to generate primary key values, unless you have a reliable mechanism that purges old rows faster than the sequence cycles.

For more information, see the "`CREATE SEQUENCE`" section in the *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

Note: The sequence is not tied to a table. Generally, you should name the sequence after its intended use. However, the sequence can be used anywhere, regardless of its name.

NEXTVAL and CURRVAL Pseudocolumns

- NEXTVAL returns the next available sequence value. It returns a unique value every time it is referenced, even for different users.
- CURRVAL obtains the current sequence value.
- NEXTVAL must be issued for that sequence before CURRVAL contains a value.

After you create your sequence, it generates sequential numbers for use in your tables. Reference the sequence values by using the NEXTVAL and CURRVAL pseudocolumns.

The NEXTVAL pseudocolumn is used to extract successive sequence numbers from a specified sequence. You must qualify NEXTVAL with the sequence name. When you reference *sequence*.NEXTVAL, a new sequence number is generated and the current sequence number is placed in CURRVAL.

The CURRVAL pseudocolumn is used to refer to a sequence number that the current user has just generated. However, NEXTVAL must be used to generate a sequence number in the current user's session before CURRVAL can be referenced. You must qualify CURRVAL with the sequence name. When you reference *sequence*.CURRVAL, the last value returned to that user's process is displayed.

Rules for Using `NEXTVAL` and `CURRVAL`

You can use `NEXTVAL` and `CURRVAL` in the following contexts:

- The `SELECT` list of a `SELECT` statement that is not part of a subquery
- The `SELECT` list of a subquery in an `INSERT` statement
- The `VALUES` clause of an `INSERT` statement
- The `SET` clause of an `UPDATE` statement

You cannot use `NEXTVAL` and `CURRVAL` in the following contexts:

- The `SELECT` list of a view
- A `SELECT` statement with the `DISTINCT` keyword
- A `SELECT` statement with `GROUP BY`, `HAVING`, or `ORDER BY` clauses
- A subquery in a `SELECT`, `DELETE`, or `UPDATE` statement

For more information, see the “Pseudocolumns” and “`CREATE SEQUENCE`” sections in *Oracle Database SQL Language Reference* for Oracle Database 12*c.*

Using a Sequence

- Insert a new department named “Support” in location ID 2500:

```
INSERT INTO departments(department_id,
            department_name, location_id)
VALUES      (dept_deptid_seq.NEXTVAL,
            'Support', 2500);
```

```
1 rows inserted
```

- View the current value for the `DEPT_DEPTID_SEQ` sequence:

```
SELECT    dept_deptid_seq.CURRVAL
FROM      dual;
```


The example in the slide inserts a new department in the `DEPARTMENTS` table. It uses the `DEPT_DEPTID_SEQ` sequence to generate a new department number as follows.

You can view the current value of the sequence using the *sequence_name*.`CURRVAL`, as shown in the second example in the slide.

Suppose that you now want to hire employees to staff the new department. The `INSERT` statement to be executed for all new employees can include the following code:

```
INSERT INTO employees (employee_id, department_id, ...)
VALUES (employees_seq.NEXTVAL, dept_deptid_seq .CURRVAL, ...);
```

Note: The preceding example assumes that a sequence called `EMPLOYEE_SEQ` has already been created to generate new employee numbers.

SQL Column Defaulting Using a Sequence

- SQL syntax for column defaults allow <sequence>.nextval, <sequence>.currval as a SQL column defaulting expression for numeric columns, where <sequence> is an Oracle database sequence.
- The DEFAULT expression can include the sequence pseudocolumns CURRVAL and NEXTVAL, as long as the sequence exists and you have the privileges necessary to access it.

```
CREATE SEQUENCE s1 START WITH 1;
CREATE TABLE emp (a1 NUMBER DEFAULT s1.NEXTVAL NOT
NULL, a2 VARCHAR2(10));
INSERT INTO emp (a2) VALUES ('john');
INSERT INTO emp (a2) VALUES ('mark');
SELECT * FROM emp;
```

```
sequence S1 created.
table EMP created.
1 rows inserted.
1 rows inserted.
A1 A2
-- ----------
 1 john
 2 mark
```


SQL syntax for column defaults has been enhanced so that it allows <sequence>.nextval, <sequence>.currval as a SQL column defaulting expression for numeric columns, where <sequence> is an Oracle database sequence.

The DEFAULT expression can include the sequence pseudocolumns CURRVAL and NEXTVAL, as long as the sequence exists and you have the privileges necessary to access it. The user inserting into a table must have access privileges to the sequence. If the sequence is dropped, subsequent insert DMLs where *expr* is used for defaulting will result in a compilation error.

In the slide example, sequence s1 is created, which starts from 1.

Caching Sequence Values

- Caching sequence values in memory gives faster access to those values.
- Gaps in sequence values can occur when:
  - A rollback occurs
  - The system crashes
  - A sequence is used in another table

You can cache sequences in memory to provide faster access to those sequence values. The cache is populated the first time you refer to the sequence. Each request for the next sequence value is retrieved from the cached sequence. After the last sequence value is used, the next request for the sequence pulls another cache of sequences into memory.

Gaps in the Sequence

Although sequence generators issue sequential numbers without gaps, this action occurs independently of a commit or rollback. Therefore, if you roll back a statement containing a sequence, the number is lost.

Another event that can cause gaps in the sequence is a system crash. If the sequence caches values in memory, those values are lost if the system crashes.

Because sequences are not tied directly to tables, the same sequence can be used for multiple tables. However, if you do so, each table can contain gaps in the sequential numbers.

Modifying a Sequence

Change the increment value, maximum value, minimum value, cycle option, or cache option:

```
ALTER SEQUENCE dept_deptid_seq
               INCREMENT BY 20
               MAXVALUE 999999
               NOCACHE
               NOCYCLE;
```

```
sequence DEPT_DEPTID_SEQ altered.
```


If you reach the `MAXVALUE` limit for your sequence, no additional values from the sequence are allocated and you will receive an error indicating that the sequence exceeds the `MAXVALUE`. To continue to use the sequence, you can modify it by using the `ALTER SEQUENCE` statement.

Syntax

```
ALTER  SEQUENCE   sequence
      [INCREMENT BY n]
      [{MAXVALUE n | NOMAXVALUE}]
      [{MINVALUE n | NOMINVALUE}]
      [{CYCLE | NOCYCLE}]
      [{CACHE n | NOCACHE}];
```

In the syntax, *`sequence`* is the name of the sequence generator.

For more information, see the section on "`ALTER SEQUENCE`" in *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

Guidelines for Modifying a Sequence

- You must be the owner or have the `ALTER` privilege for the sequence.
- Only future sequence numbers are affected.
- The sequence must be dropped and re-created to restart the sequence at a different number.
- Some validation is performed.
- To remove a sequence, use the `DROP` statement:

```
DROP SEQUENCE dept_deptid_seq;
```

```
sequence DEPT_DEPTID_SEQ dropped.
```


- You must be the owner or have the `ALTER` privilege for the sequence to modify it. You must be the owner or have the `DROP ANY SEQUENCE` privilege to remove it.
- Only future sequence numbers are affected by the `ALTER SEQUENCE` statement.
- The `START WITH` option cannot be changed using `ALTER SEQUENCE`. The sequence must be dropped and re-created to restart the sequence at a different number.
- Some validation is performed. For example, a new `MAXVALUE` that is less than the current sequence number cannot be imposed.

```
ALTER SEQUENCE dept_deptid_seq
      INCREMENT BY 20
      MAXVALUE 90
      NOCACHE
      NOCYCLE;
```

- The error:

```
SQL Error: ORA-04009: MAXVALUE cannot be made to be less than the current value
04009. 00000 - "MAXVALUE cannot be made to be less than the current value"
*Cause:    the current value exceeds the given MAXVALUE
*Action:   make sure that the new MAXVALUE is larger than the current value
```

Sequence Information

- The `USER_SEQUENCES` view describes all sequences that you own.

```
DESCRIBE user_sequences
```

```
DESCRIBE user_sequences
Name            Null     Type
--------------- -------- -------------
SEQUENCE_NAME   NOT NULL VARCHAR2(128)
MIN_VALUE                NUMBER
MAX_VALUE                NUMBER
INCREMENT_BY    NOT NULL NUMBER
CYCLE_FLAG               VARCHAR2(1)
ORDER_FLAG               VARCHAR2(1)
CACHE_SIZE      NOT NULL NUMBER
LAST_NUMBER     NOT NULL NUMBER
PARTITION_COUNT          NUMBER
SESSION_FLAG             VARCHAR2(1)
KEEP_VALUE               VARCHAR2(1)
```

- Verify your sequence values in the `USER_SEQUENCES` data dictionary table.

```
SELECT   sequence_name, min_value, max_value,
         increment_by, last_number
FROM     user_sequences;
```


The `USER_SEQUENCES` view describes all sequences that you own. When you create the sequence, you specify criteria that are stored in the `USER_SEQUENCES` view. The columns in this view are:

- `SEQUENCE_NAME`: Name of the sequence
- `MIN_VALUE`: Minimum value of the sequence
- `MAX_VALUE`: Maximum value of the sequence
- `INCREMENT_BY`: Value by which the sequence is incremented
- `CYCLE_FLAG`: Whether sequence wraps around on reaching the limit
- `ORDER_FLAG`: Whether sequence numbers are generated in order
- `CACHE_SIZE`: Number of sequence numbers to cache
- `LAST_NUMBER`: Last sequence number written to disk. If a sequence uses caching, the number written to disk is the last number placed in the sequence cache. This number is likely to be greater than the last sequence number that was used. The `LAST_NUMBER` column displays the next available sequence number if `NOCACHE` is specified.

After creating your sequence, it is documented in the data dictionary. Because a sequence is a database object, you can identify it in the `USER_OBJECTS` data dictionary table.

You can also confirm the settings of the sequence by selecting from the `USER_SEQUENCES` data dictionary view.

Lesson Agenda

- Overview of sequences:
  - Creating, using, and modifying a sequence
  - Cache sequence values
  - NEXTVAL and CURRVAL pseudocolumns
  - SQL column defaulting using a sequence
- Overview of synonyms
  - Creating, dropping synonyms
- Overview of indexes
  - Creating indexes
  - Using the CREATE TABLE statement
  - Creating function-based indexes
  - Creating multiple indexes on the same set of columns
  - Removing indexes

ORACLE

Synonyms

A synonym:

- Is a database object
- Can be created to give an alternative name to a table or to an other database object
- Requires no storage other than its definition in the data dictionary
- Is useful for hiding the identity and location of an underlying schema object

Synonyms are database object that enable you to call a table by another name.

You can create synonyms to give an alternative name to a table or to an other database object. For example, you can create a synonym for a table or view, sequence, PL/SQL program unit, user-defined object type, or another synonym.

Because a synonym is simply an alias, it requires no storage other than its definition in the data dictionary.

Synonyms can simplify SQL statements for database users. Synonyms are also useful for hiding the identity and location of an underlying schema object.

Creating a Synonym for an Object

Simplify access to objects by creating a synonym (another name for an object). With synonyms, you can:

- Create an easier reference to a table that is owned by another user
- Shorten lengthy object names

```
CREATE [PUBLIC] SYNONYM synonym
FOR    object;
```


To refer to a table that is owned by another user, you need to prefix the table name with the name of the user who created it, followed by a period. Creating a synonym eliminates the need to qualify the object name with the schema and provides you with an alternative name for a table, view, sequence, procedure, or other objects. This method can be especially useful with lengthy object names, such as views.

In the syntax:

`PUBLIC`	Creates a synonym that is accessible to all users
`synonym`	Is the name of the synonym to be created
`object`	Identifies the object for which the synonym is created

Guidelines

- The object cannot be contained in a package.
- A private synonym name must be distinct from all other objects that are owned by the same user.
- To create a `PUBLIC` synonym, you must have the `CREATE PUBLIC SYNONYM` system privilege.

For more information, see the section on "`CREATE SYNONYM`" in *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

Creating and Removing Synonyms

- Create a shortened name for the `DEPT_SUM_VU` view:

```
CREATE SYNONYM  d_sum
FOR  dept_sum_vu;
```

```
synonym D_SUM created.
```

- Drop a synonym:

```
DROP SYNONYM d_sum;
```

```
synonym D_SUM dropped.
```


Creating a Synonym

The slide example creates a synonym for the `DEPT_SUM_VU` view for quicker reference.

The database administrator can create a public synonym that is accessible to all users. The following example creates a public synonym named `DEPT` for Alice's `DEPARTMENTS` table:

```
CREATE PUBLIC SYNONYM  dept
FOR    alice.departments;
```

Removing a Synonym

To remove a synonym, use the `DROP SYNONYM` statement. Only the database administrator can drop a public synonym.

```
DROP PUBLIC SYNONYM  dept;
```

For more information, see the section on "`DROP SYNONYM`" in *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

Synonym Information

```
DESCRIBE user_synonyms
```

```
DESCRIBE user_synonyms
Name         Null     Type
------------ -------- -------------
SYNONYM_NAME NOT NULL VARCHAR2(128)
TABLE_OWNER           VARCHAR2(128)
TABLE_NAME   NOT NULL VARCHAR2(128)
DB_LINK               VARCHAR2(128)
```

```
SELECT *
FROM   user_synonyms;
```

	SYNONYM_NAME	TABLE_OWNER	TABLE_NAME	DB_LINK
1	D_SUM	ORA21	DEPT_SUM_VU	(null)

The USER_SYNONYMS dictionary view describes private synonyms (synonyms that you own).

You can query this view to find your synonyms. You can query ALL_SYNONYMS to find out the name of all the synonyms that are available to you and the objects on which these synonyms apply.

The columns in this view are:

- SYNONYM_NAME: Name of the synonym
- TABLE_OWNER: Owner of the object that is referenced by the synonym
- TABLE_NAME: Name of the table or view that is referenced by the synonym
- DB_LINK: Name of the database link reference (if any)

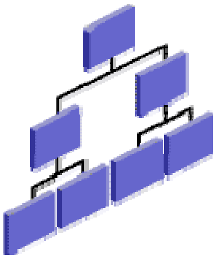
Lesson Agenda

- Overview of sequences:
  - Creating, using, and modifying a sequence
  - Cache sequence values
  - `NEXTVAL` and `CURRVAL` pseudocolumns
  - SQL column defaulting using a sequence
- Overview of synonyms
  - Creating, dropping synonyms
- Overview of indexes
  - Creating indexes
  - Using the `CREATE TABLE` statement
  - Creating function-based indexes
  - Creating multiple indexes on the same set of columns
  - Removing indexes

Indexes

An index:

- Is a schema object
- Can be used by the Oracle Server to speed up the retrieval of rows by using a pointer
- Can reduce disk input/output (I/O) by using a rapid path access method to locate data quickly
- Is dependent on the table that it indexes
- Is used and maintained automatically by the Oracle Server



An Oracle Server index is a schema object that can speed up the retrieval of rows by using a pointer and improves the performance of some queries. Indexes can be created explicitly or automatically. If you do not have an index on the column, a full table scan occurs.

An index provides direct and fast access to rows in a table. Its purpose is to reduce the disk I/O by using an indexed path to locate data quickly. An index is used and maintained automatically by the Oracle Server. After an index is created, no direct activity is required by the user.

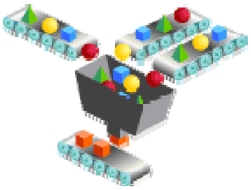
Indexes are logically and physically independent of the data in the objects with which they are associated. This means that they can be created or dropped at any time, and have no effect on the base tables or other indexes.

Note: When you drop a table, the corresponding indexes are also dropped.

For more information, see the section on “Schema Objects: Indexes” in *Oracle Database Concepts* 12*c Release* 1.

How Are Indexes Created?

- Automatically: A unique index is created automatically when you define a `PRIMARY KEY` or `UNIQUE` constraint in a table definition.



- Manually: You can create unique or nonunique index on columns to speed up access to the rows.

You can create two types of indexes.

- **Unique index:** The Oracle Server automatically creates this index when you define a column in a table to have a `PRIMARY KEY` or a `UNIQUE` constraint. The name of the index is the name that is given to the constraint.
- **Nonunique index:** This is an index that a user can create. For example, you can create the `FOREIGN KEY` column index for a join in a query to improve the speed of retrieval.

Note: You can manually create a unique index, but it is recommended that you create a unique constraint, which implicitly creates a unique index.

Creating an Index

- Create an index on one or more columns:

```
CREATE [UNIQUE]INDEX index
ON table (column[, column]...);
```

- Improve the speed of query access to the LAST_NAME column in the EMPLOYEES table:

```
CREATE INDEX emp_last_name_idx
ON employees(last_name);
```

```
index EMP_LAST_NAME_IDX created.
```

ORACLE

Create an index on one or more columns by issuing the CREATE INDEX statement.

In the syntax:

- `index` Is the name of the index
- `table` Is the name of the table
- `Column` Is the name of the column in the table to be indexed

Specify UNIQUE to indicate that the value of the column (or columns) upon which the index is based must be unique. Specify BITMAP to indicate that the index is to be created with a bitmap for each distinct key, rather than indexing each row separately. Bitmap indexes store the rowids associated with a key value as a bitmap.

For more information, see the section on "CREATE INDEX" in *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

CREATE INDEX with the CREATE TABLE Statement

```
CREATE TABLE NEW_EMP
(employee_id NUMBER(6)
                  PRIMARY KEY USING INDEX
                 (CREATE INDEX emp_id_idx ON
                 NEW_EMP(employee_id)),
first_name   VARCHAR2(20),
last_name    VARCHAR2(25));
```

```
table NEW_EMP created.
```

```
SELECT INDEX_NAME, TABLE_NAME
FROM   USER_INDEXES
WHERE  TABLE_NAME = 'NEW_EMP';
```

	INDEX_NAME	TABLE_NAME
1	EMP_ID_IDX	NEW_EMP

In the example in the slide, the CREATE INDEX clause is used with the CREATE TABLE statement to create a PRIMARY KEY index explicitly. You can name your indexes at the time of PRIMARY KEY creation to be different from the name of the PRIMARY KEY constraint.

You can query the USER_INDEXES data dictionary view for information about your indexes.

The following example illustrates the database behavior if the index is not explicitly named:

```
CREATE TABLE EMP_UNNAMED_INDEX
   (employee_id NUMBER(6) PRIMARY KEY ,
    first_name VARCHAR2(20),
    last_name VARCHAR2(25));
```

```
SELECT INDEX_NAME, TABLE_NAME
FROM   USER_INDEXES
WHERE  TABLE_NAME = 'EMP_UNNAMED_INDEX';
```

Observe that the Oracle Server gives a generic name to the index that is created for the `PRIMARY KEY` column.

You can also use an existing index for your `PRIMARY KEY` column—for example, when you are expecting a large data load and want to speed up the operation. You may want to disable the constraints while performing the load and then enable them, in which case having a unique index on the `PRIMARY KEY` will still cause the data to be verified during the load. Therefore, you can first create a nonunique index on the column designated as `PRIMARY KEY`, and then create the `PRIMARY KEY` column and specify that it should use the existing index. The following examples illustrate this process:

Step 1: Create the table:

```
CREATE TABLE NEW_EMP2
 (employee_id NUMBER(6),
 first_name  VARCHAR2(20),
 last_name   VARCHAR2(25)
 );
```

Step 2: Create the index:

```
CREATE INDEX emp_id_idx2 ON
  new_emp2(employee_id);
```

Step 3: Create the `PRIMARY KEY`:

```
ALTER TABLE new_emp2 ADD PRIMARY KEY (employee_id) USING INDEX
 emp_id_idx2;
```

Function-Based Indexes

- A function-based index is based on expressions.
- The index expression is built from table columns, constants, SQL functions, and user-defined functions.

```
CREATE INDEX upper_dept_name_idx
ON dept2(UPPER(department_name));
```

```
index UPPER_DEPT_NAME_IDX created.
```

```
SELECT *
FROM   dept2
WHERE  UPPER(department_name) = 'SALES';
```

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	80	Sales	145	2500

Function-based indexes defined with the `UPPER(`*`column_name`*`)` or `LOWER(`*`column_name`*`)` keywords allow non-case-sensitive searches. For example, consider the following index:

```
CREATE INDEX upper_last_name_idx ON emp2 (UPPER(last_name));
```

This facilitates processing queries such as:

```
SELECT * FROM emp2 WHERE UPPER(last_name) = 'KING';
```

The Oracle Server uses the index only when that particular function is used in a query. For example, the following statement may use the index, but without the `WHERE` clause, the Oracle Server may perform a full table scan:

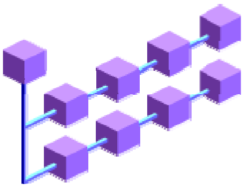
```
SELECT    *
FROM      employees
WHERE     UPPER (last_name) IS NOT NULL
ORDER BY UPPER (last_name);
```

Note: For creating a function-based index, you need the `QUERY REWRITE` system privilege. The `QUERY_REWRITE_ENABLED` initialization parameter must be set to `TRUE` for a function-based index to be used.

The Oracle Server treats indexes with columns marked `DESC` as function-based indexes. The columns marked `DESC` are sorted in descending order.

Creating Multiple Indexes on the Same Set of Columns

- You can create multiple indexes on the same set of columns.
- Multiple indexes can be created on the same set of columns if:
  - The indexes are of different types
  - The indexes uses different partitioning
  - The indexes have different uniqueness properties



You can create multiple indexes on the same set of columns if the indexes are of different types, use different partitioning, or have different uniqueness properties. For example, you can create a B-tree index and a bitmap index on the same set of columns.

Similarly, you can create both a unique and non-unique index on the same set of columns.

When you have multiple indexes on the same set of columns, only one of these indexes can be visible at a time.

Example of Creating Multiple Indexes on the Same Set of Columns

```
CREATE INDEX emp_id_name_ix1
ON employees(employee_id, first_name);
```

```
index EMP_ID_NAME_IX1 created.
```

```
ALTER INDEX emp_id_name_ix1 INVISIBLE;
```

```
index EMP_ID_NAME_IX1 altered.
```

```
CREATE BITMAP INDEX emp_id_name_ix2
ON employees(employee_id, first_name);
```

```
bitmap index EMP_ID_NAME_IX2 created.
```


The code example shows the creation of a B-tree index, `emp_id_name_ix1`, on the `employee_id` and `first_name` column of the employees table in the `HR` schema. After the creation of the index, it is altered to make it invisible. Then a bitmap index is created on the `employee_id` and `first_name` column of the `employees` table in the `HR` schema. The bitmap index, `emp_id_name_ix2`, is visible by default.

Index Information

- `USER_INDEXES` provides information about your indexes.
- `USER_IND_COLUMNS` describes columns of indexes owned by you and columns of indexes on your tables.

```
DESCRIBE user_indexes
```

```
DESCRIBE user_indexes
Name                    Null     Type
----------------------- -------- --------------
INDEX_NAME              NOT NULL VARCHAR2(128)
INDEX_TYPE                       VARCHAR2(27)
TABLE_OWNER             NOT NULL VARCHAR2(128)
TABLE_NAME              NOT NULL VARCHAR2(128)
TABLE_TYPE                       VARCHAR2(11)
UNIQUENESS                       VARCHAR2(9)
```

...

You query the `USER_INDEXES` view to find out the names of your indexes, the table name on which the index is created, and whether the index is unique.

Note: For a complete listing and description of the columns in the `USER_INDEXES` view, see "`USER_INDEXES`" in the *Oracle® Database Reference 12c Release 1*.

USER_INDEXES: Examples

a

```
SELECT index_name, table_name, uniqueness
FROM   user_indexes
WHERE  table_name = 'EMPLOYEES';
```

	INDEX_NAME	TABLE_NAME	UNIQUENESS
1	EMP_NAME_IX	EMPLOYEES	NONUNIQUE
2	EMP_MANAGER_IX	EMPLOYEES	NONUNIQUE
3	EMP_JOB_IX	EMPLOYEES	NONUNIQUE
4	EMP_DEPARTMENT_IX	EMPLOYEES	NONUNIQUE
5	EMP_EMP_ID_PK	EMPLOYEES	UNIQUE
6	EMP_EMAIL_UK	EMPLOYEES	UNIQUE

…

b

```
SELECT index_name, table_name
FROM   user_indexes
WHERE  table_name = 'EMP_LIB';
```

	INDEX_NAME	TABLE_NAME
1	SYS_C0010979	EMP_LIB

In slide example **a**, the USER_INDEXES view is queried to find the name of the index, name of the table on which the index is created, and whether the index is unique.

In slide example **b**, observe that the Oracle Server gives a generic name to the index that is created for the PRIMARY KEY column. The EMP_LIB table is created by using the following code:

```
CREATE TABLE emp_lib
   (book_id NUMBER(6) PRIMARY KEY,
    title VARCHAR2(25),
    category VARCHAR2(20));
```

Querying USER_IND_COLUMNS

```
DESCRIBE user_ind_columns
```

```
DESCRIBE user_ind_columns
Name            Null Type
--------------- ---- --------------
INDEX_NAME           VARCHAR2(128)
TABLE_NAME           VARCHAR2(128)
COLUMN_NAME          VARCHAR2(4000)
COLUMN_POSITION      NUMBER
COLUMN_LENGTH        NUMBER
CHAR_LENGTH          NUMBER
DESCEND              VARCHAR2(4)
```

```
SELECT index_name, column_name,table_name
FROM   user_ind_columns
WHERE  index_name = 'LNAME_IDX';
```

	INDEX_NAME	COLUMN_NAME	TABLE_NAME
1	LNAME_IDX	LAST_NAME	EMP_TEST

ORACLE

The USER_IND_COLUMNS dictionary view provides information such as the name of the index, name of the indexed table, name of a column within the index, and the column's position within the index.

For the slide example, the emp_test table and LNAME_IDX index are created by using the following code:

```
CREATE TABLE emp_test AS SELECT * FROM employees;
CREATE INDEX lname_idx ON emp_test(last_name);
```

Removing an Index

- Remove an index from the data dictionary by using the `DROP INDEX` command:

```
DROP INDEX index;
```

- Remove the `emp_last_name_idx` index from the data dictionary:

```
DROP INDEX emp_last_name_idx;
```

```
index EMP_LAST_NAME_IDX dropped.
```

- To drop an index, you must be the owner of the index or have the `DROP ANY INDEX` privilege.

You cannot modify indexes. To change an index, you must drop it and then re-create it.

Remove an index definition from the data dictionary by issuing the `DROP INDEX` statement. To drop an index, you must be the owner of the index or have the `DROP ANY INDEX` privilege.

In the syntax, *`index`* is the name of the index.

You can drop an index using the `ONLINE` keyword.

```
DROP INDEX emp_indx ONLINE;
```

`ONLINE`: Specify `ONLINE` to indicate that DML operations on the table are allowed while dropping the index.

Note: If you drop a table, indexes and constraints are automatically dropped but views remain.

Quiz

Indexes must be created manually and serve to speed up access to rows in a table.

a. True

b. False

Answer: b

Note: Indexes are designed to speed up query performance. However, not all indexes are created manually. The Oracle server automatically creates an index when you define a column in a table to have a `PRIMARY KEY` or a `UNIQUE` constraint.

Summary

In this lesson, you should have learned how to:

- Automatically generate sequence numbers by using a sequence generator
- Use synonyms to provide alternative names for objects
- Create indexes to improve the speed of query retrieval
- Find information about your objects through the following dictionary views:
  - `USER_VIEWS`
  - `USER_SEQUENCES`
  - `USER_SYNONYMS`

In this lesson, you should have learned about database objects such as sequences, indexes, and synonyms.

Practice 3: Overview

This practice covers the following topics:

- Creating sequences
- Using sequences
- Querying the dictionary views for sequence information
- Creating synonyms
- Querying the dictionary views for synonyms information
- Creating indexes
- Querying the dictionary views for indexes information

This lesson's practice provides you with a variety of exercises in creating and using a sequence, an index, and a synonym. You also learn how to query the data dictionary views for sequence, synonyms, and index information.

4

Creating Views

Objectives

After completing this lesson, you should be able to do:

- Create simple and complex views
- Retrieve data from views
- Querying the dictionary views for the view information

In this lesson, you are introduced to views, and you learn the basics of creating and using views.

Lesson Agenda

- Overview of views
- Creating, modifying, and retrieving data from a view
- Data Manipulation Language (DML) operations on a view
- Dropping a view

Database Objects

Object	Description
Table	Basic unit of storage; composed of rows
View	Logically represents subsets of data from one or more tables
Sequence	Generates numeric values
Index	Improves the performance of data retrieval queries
Synonym	Gives alternative names to objects

There are several other objects in a database in addition to tables.

With views, you can present and hide data from the tables.

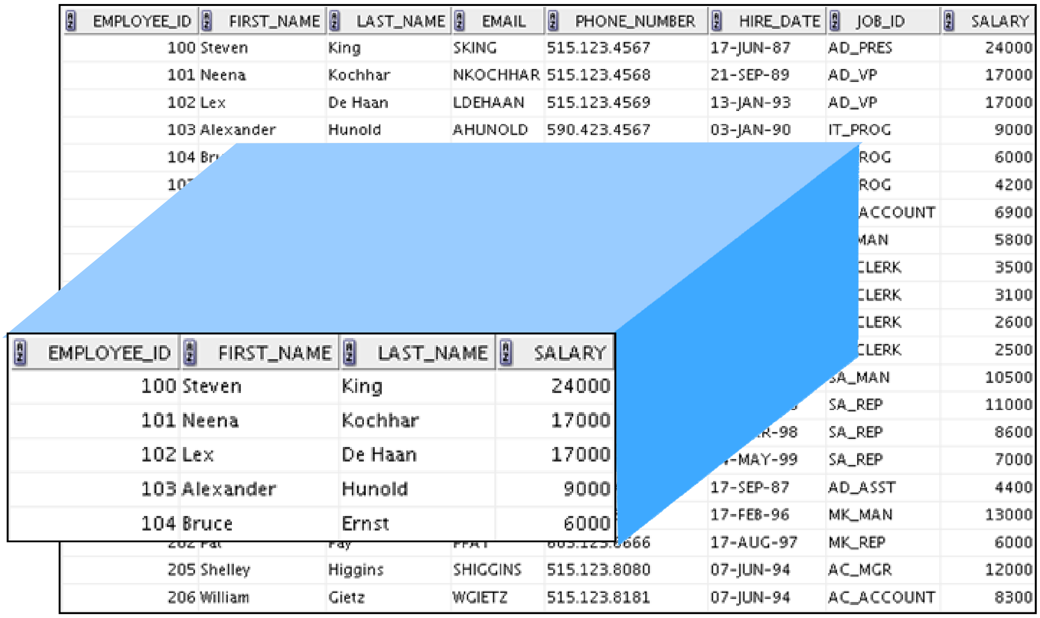
Many applications require the use of unique numbers as primary key values. You can either build code into the application to handle this requirement or use a sequence to generate unique numbers.

If you want to improve the performance of data retrieval queries, you should consider creating an index. You can also use indexes to enforce uniqueness on a column or a collection of columns.

You can provide alternative names for objects by using synonyms.

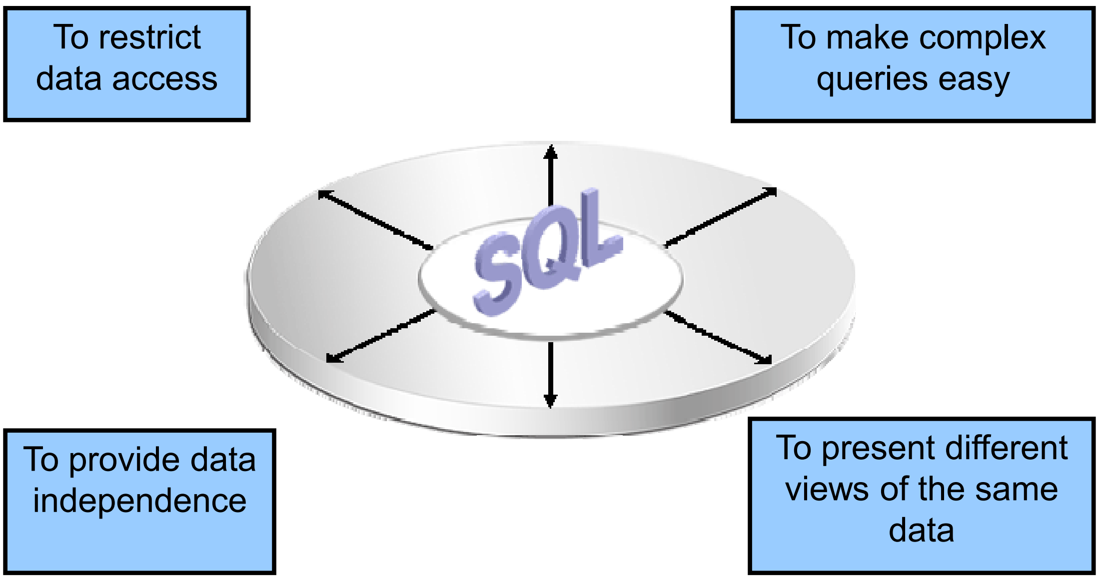
What Is a View?

`EMPLOYEES` table



You can present logical subsets or combinations of data by creating views of tables. A view is a schema object , a stored `SELECT` statement based on a table or another view. A view contains no data of its own, but is like a window through which data from tables can be viewed or changed. The tables on which a view is based are called *base tables*. The view is stored as a `SELECT` statement in the data dictionary.

Advantages of Views



- Views restrict access to the data because they display selected columns from the table.
- Views can be used to make simple queries to retrieve the results of complicated queries. For example, views can be used to query information from multiple tables without the user knowing how to write a join statement.
- Views provide data independence for ad hoc users and application programs. One view can be used to retrieve data from several tables.
- Views provide groups of users access to data according to their particular criteria.

For more information, see the "`CREATE VIEW`" section in *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

Simple Views and Complex Views

Feature	Simple Views	Complex Views
Number of tables	One	One or more
Contain functions	No	Yes
Contain groups of data	No	Yes
DML operations through a view	Yes	Not always

There are two classifications for views: simple and complex. The basic difference is related to the DML (`INSERT`, `UPDATE`, and `DELETE`) operations.

- A simple view is one that:
  - Derives data from only one table
  - Contains no functions or groups of data
  - Usually perform DML operations through the view
- A complex view is one that:
  - Derives data from many tables
  - Contains functions or groups of data
  - Does not always allow DML operations through the view

Lesson Agenda

- Overview of views
- Creating, modifying, and retrieving data from a view
- Data Manipulation Language (DML) operations on a view
- Dropping a view

Creating a View

- You embed a subquery in the CREATE VIEW statement:

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view
  [(alias[, alias]...)]
 AS subquery
[WITH CHECK OPTION [CONSTRAINT constraint]]
[WITH READ ONLY [CONSTRAINT constraint]];
```

- The subquery can contain complex SELECT syntax.

You can create a view by embedding a subquery in the CREATE VIEW statement.

In the syntax:

OR REPLACE	Re-creates the view if it already exists. You can use this clause to change the definition of an existing view without dropping, re-creating, and regranting object privileges previously granted on it.
FORCE	Creates the view regardless of whether or not the base tables exist
NOFORCE	Creates the view only if the base tables exist (This is the default.)
view	Is the name of the view
alias	Specifies names for the expressions selected by the view's query (The number of aliases must match the number of expressions selected by the view.)
subquery	Is a complete SELECT statement (You can use aliases for the columns in the SELECT list.)
WITH CHECK OPTION	Specifies that only those rows that are accessible to the view can be inserted or updated
Constraint	Is the name assigned to the CHECK OPTION constraint
WITH READ ONLY	Ensures that no DML operations can be performed on this view

Note: In SQL Developer, click the Run Script icon or press F5 to run the data definition language (DDL) statements. The feedback messages will be shown on the Script Output tabbed page.

Creating a View

- Create the `EMPVU80` view, which contains details of the employees in department 80:

```
CREATE VIEW   empvu80
 AS SELECT  employee_id, last_name, salary
    FROM    employees
    WHERE   department_id = 80;
```

```
view EMPVU80 created.
```

- Describe the structure of the view by using the SQL*Plus `DESCRIBE` command:

```
DESCRIBE empvu80;
```


The example in the slide creates a view that contains the employee number, last name, and salary for each employee in department 80.

You can display the structure of the view by using the `DESCRIBE` command.

```
DESCRIBE empvu80
Name        Null     Type
----------- -------- ------------
EMPLOYEE_ID NOT NULL NUMBER(6)
LAST_NAME   NOT NULL VARCHAR2(25)
SALARY               NUMBER(8,2)
```

Guidelines

- The subquery that defines a view can contain complex `SELECT` syntax, including joins, groups, and subqueries.
- If you do not specify a constraint name for the view created with the `WITH CHECK OPTION`, the system assigns a default name in the `SYS_Cn` format.
- You can use the `OR REPLACE` option to change the definition of the view without dropping and re-creating it, or regranting the object privileges previously granted on it.

Creating a View

- Create a view by using column aliases in the subquery:

```
CREATE VIEW   salvu50
 AS SELECT  employee_id ID_NUMBER, last_name NAME,
            salary*12 ANN_SALARY
    FROM    employees
    WHERE   department_id = 50;
```

```
view SALVU50 created.
```

- Select the columns from this view by the given alias names.

ORACLE

You can control the column names by including column aliases in the subquery.

The example in the slide creates a view containing the employee number (`EMPLOYEE_ID`) with the alias `ID_NUMBER`, name (`LAST_NAME`) with the alias `NAME`, and annual salary (`SALARY`) with the alias `ANN_SALARY` for every employee in department 50.

Alternatively, you can use an alias after the `CREATE` statement and before the `SELECT` subquery. The number of aliases listed must match the number of expressions selected in the subquery.

```
CREATE OR REPLACE VIEW salvu50 (ID_NUMBER, NAME, ANN_SALARY)
  AS SELECT  employee_id, last_name, salary*12
     FROM    employees
     WHERE   department_id = 50;
```

Retrieving Data from a View

```
SELECT *
FROM   salvu50;
```

	ID_NUMBER	NAME	ANN_SALARY
1	120	Weiss	96000
2	121	Fripp	98400
3	122	Kaufling	94800
4	123	Vollman	78000
5	124	Mourgos	69600
6	125	Nayer	38400
7	126	Mikkilineni	32400

…

You can retrieve data from a view as you would from any table. You can display either the contents of the entire view or just specific rows and columns.

Modifying a View

- Modify the `EMPVU80` view by using a `CREATE OR REPLACE VIEW` clause. Add an alias for each column name:

```
CREATE OR REPLACE VIEW empvu80
  (id_number, name, sal, department_id)
AS SELECT  employee_id, first_name || ' '
           || last_name, salary, department_id
   FROM    employees
   WHERE   department_id = 80;
```

```
view EMPVU80 created.
```

- Column aliases in the `CREATE OR REPLACE VIEW` clause are listed in the same order as the columns in the subquery.

With the `REPLACE` option, a view can be created even if one exists with this name already, thus replacing the old version of the view for its owner. This means that the view can be altered without dropping, re-creating, and regranting object privileges.

Note: When assigning column aliases in the `CREATE OR REPLACE VIEW` clause, remember that the aliases are listed in the same order as the columns in the subquery.

Creating a Complex View

Create a complex view that contains group functions to display values from two tables:

```
CREATE OR REPLACE VIEW dept_sum_vu
  (name, minsal, maxsal, avgsal)
AS SELECT   d.department_name, MIN(e.salary),
            MAX(e.salary),AVG(e.salary)
   FROM     employees e JOIN departments d
   USING    (department_id)
   GROUP BY d.department_name;
```

```
view DEPT_SUM_VU created.
```


The example in the slide creates a complex view of department names, minimum salaries, maximum salaries, and the average salaries by department. Note that alternative names have been specified for the view. This is a requirement if any column of the view is derived from a function or an expression.

You can view the structure of the view by using the `DESCRIBE` command. Display the contents of the view by issuing a `SELECT` statement.

```
SELECT  *
FROM    dept_sum_vu;
```

View Information

1

```
DESCRIBE user_views
```

```
Name                            Null     Type
------------------------------- -------- ------------
VIEW_NAME                       NOT NULL VARCHAR2(30)
TEXT_LENGTH                              NUMBER
TEXT                                     LONG()
```

...

2

```
SELECT view_name FROM user_views;
```

	VIEW_NAME
1	EMP_DETAILS_VIEW
2	SALVU50
3	EMPVU80
4	DEPT_SUM_VU

3

```
SELECT text FROM user_views
WHERE view_name = 'EMP_DETAILS_VIEW';
```

	TEXT
1	SELECT e.employee_id, e.job_id, e.manager_id, e.department_id, d.location_id, l.co

...

```
AND c.region_id = r.region_id AND j.job_id = e.job_idWITH READ ONLY
```

ORACLE

After your view is created, you can query the data dictionary view called `USER_VIEWS` to see the name of the view and the view definition. The text of the `SELECT` statement that constitutes your view is stored in a `LONG` column. The `TEXT_LENGTH` column is the number of characters in the `SELECT` statement. By default, when you select from a `LONG` column, only the first 80 characters of the column's value are displayed. To see more than 80 characters in SQL*Plus, use the `SET LONG` command:

```
SET LONG 1000
```

In the examples in the slide:

1. The `USER_VIEWS` columns are displayed. Note that this is a partial listing.
2. The names of your views are retrieved
3. The `SELECT` statement for the `EMP_DETAILS_VIEW` is displayed from the dictionary

Data Access Using Views

When you access data by using a view, the Oracle Server performs the following operations:

- It retrieves the view definition from the data dictionary table `USER_VIEWS`.
- It checks access privileges for the view base table.
- It converts the view query into an equivalent operation on the underlying base table or tables. That is, data is retrieved from, or an update is made to, the base tables.

Lesson Agenda

- Overview of views
- Creating, modifying, and retrieving data from a view
- Data Manipulation Language (DML) operations on a view
- Dropping a view

Rules for Performing DML Operations on a View

- You can usually perform DML operations on simple views.
- You cannot remove a row if the view contains the following:
  - Group functions
  - A `GROUP BY` clause
  - The `DISTINCT` keyword
  - The pseudocolumn `ROWNUM` keyword

- You can perform DML operations on data through a view if those operations follow certain rules.
- You can remove a row from a view unless it contains any of the following:
  - Group functions
  - A `GROUP BY` clause
  - The `DISTINCT` keyword
  - The pseudocolumn `ROWNUM` keyword

Rules for Performing DML Operations on a View

You cannot modify data in a view if it contains:

- Group functions
- A `GROUP BY` clause
- The `DISTINCT` keyword
- The pseudocolumn `ROWNUM` keyword
- Expressions

You can modify data through a view unless it contains any of the conditions mentioned in the previous slide.

Rules for Performing DML Operations on a View

You cannot add data through a view if the view includes:

- Group functions
- A `GROUP BY` clause
- The `DISTINCT` keyword
- The pseudocolumn `ROWNUM` keyword
- Columns defined by expressions
- `NOT NULL` columns without default value in the base tables that are not selected by the view

You can add data through a view unless it contains any of the items listed in the slide. You cannot add data to a view if the view contains `NOT NULL` columns without default values in the base table. All the required values must be present in the view. Remember that you are adding values directly to the underlying table *through* the view.

For more information, see the "`CREATE VIEW`" section in *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

Using the WITH CHECK OPTION Clause

- You can ensure that DML operations performed on the view stay in the domain of the view by using the WITH CHECK OPTION clause:

```
CREATE OR REPLACE VIEW empvu20
AS SELECT     *
   FROM      employees
   WHERE     department_id = 20
   WITH CHECK OPTION CONSTRAINT empvu20_ck ;
```

```
view EMPVU20 created.
```

- Any attempt to INSERT a row with a department_id other than 20 or to UPDATE the department number for any row in the view fails because it violates the WITH CHECK OPTION.

ORACLE

It is possible to perform referential integrity checks through views. You can also enforce constraints at the database level. The view can be used to protect data integrity, but the use is very limited.

The WITH CHECK OPTION clause specifies that INSERTs and UPDATEs performed through the view cannot create rows that the view cannot select. Therefore, it enables integrity constraints and data validation checks to be enforced on data being inserted or updated. If there is an attempt to perform DML operations on rows that the view has not selected, an error is displayed, along with the constraint name if that has been specified.

```
UPDATE empvu20
SET    department_id = 10
WHERE  employee_id = 201;
```

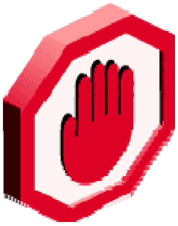
Error:

```
SQL Error: ORA-01402: view WITH CHECK OPTION where-clause violation
01402. 00000 - "view WITH CHECK OPTION where-clause violation"
*Cause:
*Action:
```

Note: No rows are updated because, if the department number were to change to 10, the view would no longer be able to see that employee. With the WITH CHECK OPTION clause, therefore, the view can see only the employees in department 20 and does not allow the department number for those employees to be changed through the view.

Denying DML Operations

- You can ensure that no DML operations occur by adding the `WITH READ ONLY` option to your view definition.
- Any attempt to perform a DML operation on any row in the view results in an Oracle server error.



You can ensure that no DML operations occur on your view by creating it with the `WITH READ ONLY` option. The example in the next slide modifies the `EMPVU10` view to prevent any DML operations on the view.

Denying DML Operations

```
CREATE OR REPLACE VIEW empvu10
    (employee_number, employee_name, job_title)
AS SELECT     employee_id, last_name, job_id
   FROM      employees
   WHERE     department_id = 10
   WITH READ ONLY ;
```

```
view EMPVU10 created.
```


Any attempt to remove a row from a view with a read-only constraint results in an error:

```
DELETE FROM empvu10
WHERE  employee_number = 200;
```

Similarly, any attempt to insert a row or modify a row using the view with a read-only constraint results in the same error.

```
Error report:
SQL Error: ORA-42399: cannot perform a DML operation on a read-only view
```

Lesson Agenda

- Overview of views
- Creating, modifying, and retrieving data from a view
- Data Manipulation Language (DML) operations on a view
- Dropping a view

Removing a View

You can remove a view without losing data because a view is based on underlying tables in the database.

```
DROP VIEW view;
```

```
DROP VIEW empvu80;
```

```
view EMPVU80 dropped.
```


You use the `DROP VIEW` statement to remove a view. The statement removes the view definition from the database. However, dropping views has no effect on the tables on which the view was based. Alternatively, views or other applications based on the deleted views become invalid. Only the creator or a user with the `DROP ANY VIEW` privilege can remove a view.

In the syntax, *`view`* is the name of the view.

Quiz

You cannot add data through a view if the view includes a `GROUP BY` clause.

a. True

b. False

Answer: a

Summary

In this lesson, you should have learned how to:

- Create, use, and remove views
- Query the dictionary views for view information

In this lesson, you should have learned about views.

Practice 4: Overview

This practice covers the following topics:

- Creating a simple view
- Creating a complex view
- Creating a view with a check constraint
- Attempting to modify data in the view
- Querying the dictionary views for view information
- Removing views

The practice provides you with a variety of exercises in creating, using, querying data dictionary views for view information, and removing views.

5

Managing Schema Objects

Objectives

After completing this lesson, you should be able to:

- Manage constraints
- Create and use temporary tables
- Create and use external tables

This lesson contains information about constraints and altering existing objects. You also learn about external tables and the provision to name the index at the time of creating a `PRIMARY KEY` constraint.

Lesson Agenda

- Managing constraints:
  - Adding and dropping a constraint
  - Enabling and disabling a constraint
  - Deferring constraints
- Creating and using temporary tables
- Creating and using external tables

Adding a Constraint Syntax

Use the `ALTER TABLE` statement to:

- Add or drop a constraint, but not to modify its structure
- Enable or disable constraints
- Add a `NOT NULL` constraint by using the `MODIFY` clause

```
ALTER TABLE  <table_name>
ADD [CONSTRAINT <constraint_name>]
type (<column_name>);
```


You can add a constraint for existing tables by using the `ALTER TABLE` statement with the `ADD` clause.

In the syntax:

table	Is the name of the table
constraint	Is the name of the constraint
type	Is the constraint type
column	Is the name of the column affected by the constraint

The constraint name syntax is optional, although recommended. If you do not name your constraints, the system generates constraint names.

Guidelines

- You can add, drop, enable, or disable a constraint, but you cannot modify its structure.
- You can add a `NOT NULL` constraint to an existing column by using the `MODIFY` clause of the `ALTER TABLE` statement.

Note: You can define a `NOT NULL` column only if the table is empty or if the column has a value for every row.

Adding a Constraint

Add a FOREIGN KEY constraint to the EMP2 table indicating that a manager must already exist as a valid employee in the EMP2 table.

```
ALTER TABLE emp2
MODIFY employee_id PRIMARY KEY;
```

```
table EMP2 altered.
```

```
ALTER TABLE emp2
ADD CONSTRAINT emp_mgr_fk
  FOREIGN KEY(manager_id)
  REFERENCES emp2(employee_id);
```

```
table EMP2 altered.
```


The first example in the slide modifies the EMP2 table to add a PRIMARY KEY constraint on the EMPLOYEE_ID column. Note that because no constraint name is provided, the constraint is automatically named by the Oracle Server. The second example in the slide creates a FOREIGN KEY constraint on the EMP2 table. The constraint ensures that a manager exists as a valid employee in the EMP2 table.

Dropping a Constraint

- The `drop_constraint_clause` enables you to drop an integrity constraint from a database.
- Remove the manager constraint from the `EMP2` table:

```
ALTER TABLE emp2
DROP CONSTRAINT emp_mgr_fk;
```

```
table EMP2 altered.
```

- Remove the `PRIMARY KEY` constraint on the `DEPT2` table and drop the associated `FOREIGN KEY` constraint on the `EMP2.DEPARTMENT_ID` column:

```
ALTER TABLE emp2
DROP PRIMARY KEY CASCADE;
```

```
table EMP2 altered.
```


The `drop_constraint_clause` enables you to drop an integrity constraint from a database.

To drop a constraint, you can identify the constraint name from the `USER_CONSTRAINTS` and `USER_CONS_COLUMNS` data dictionary views. Then use the `ALTER TABLE` statement with the `DROP` clause. The `CASCADE` option of the `DROP` clause causes any dependent constraints also to be dropped.

Syntax

```
ALTER TABLE    table
 DROP  PRIMARY KEY | UNIQUE (column) |
       CONSTRAINT   constraint   [CASCADE];
```

In the syntax:

table	Is the name of the table
column	Is the name of the column affected by the constraint
constraint	Is the name of the constraint

When you drop an integrity constraint, that constraint is no longer enforced by the Oracle Server and is no longer available in the data dictionary.

Dropping a CONSTRAINT ONLINE

You can specify the ONLINE keyword to indicate that DML operations on the table are allowed while dropping the constraint.

```
ALTER TABLE myemp2
DROP CONSTRAINT emp_name_pk ONLINE;
```


You can also drop a constraint using an ONLINE keyword.

```
ALTER TABLE myemp2
DROP CONSTRAINT emp_id_pk ONLINE;
```

Use the ALTER TABLE statement with the DROP clause. The ONLINE option of the DROP clause indicates that DML operations on the table are allowed while dropping the constraint.

ON DELETE Clause

- Use the ON DELETE CASCADE clause to delete child rows when a parent key is deleted:

```
ALTER TABLE dept2 ADD CONSTRAINT dept_lc_fk
FOREIGN KEY (location_id)
REFERENCES locations(location_id) ON DELETE CASCADE;
```

```
table DEPT2 altered.
```

- Use the ON DELETE SET NULL clause to set the child rows value to null when a parent key is deleted:

```
ALTER TABLE emp2 ADD CONSTRAINT emp_dt_fk
FOREIGN KEY (Department_id)
REFERENCES departments(department_id) ON DELETE SET NULL;
```

```
table EMP2 altered.
```

ORACLE

ON DELETE

By using the ON DELETE clause, you can determine how Oracle Database handles referential integrity if you remove a referenced primary or unique key value.

ON DELETE CASCADE

The ON DELETE CASCADE action allows parent key data that is referenced from the child table to be deleted, but not updated. When data in the parent key is deleted, all the rows in the child table that depend on the deleted parent key values are also deleted. To specify this referential action, include the ON DELETE CASCADE option in the definition of the FOREIGN KEY constraint.

ON DELETE SET NULL

When data in the parent key is deleted, the ON DELETE SET NULL action causes all the rows in the child table that depend on the deleted parent key value to be converted to null.

If you omit this clause, Oracle does not allow you to delete referenced key values in the parent table that have dependent rows in the child table.

Cascading Constraints

- The CASCADE CONSTRAINTS clause:
  - Is used along with the DROP COLUMN clause
  - Drops all referential integrity constraints that refer to the PRIMARY and UNIQUE keys defined on the dropped columns
  - Drops all multicolumn constraints defined on the dropped columns

This statement illustrates the usage of the CASCADE CONSTRAINTS clause. Assume that the TEST1 table is created as follows:

```
CREATE TABLE test1 (
  col1_pk NUMBER PRIMARY KEY,
  col2_fk NUMBER,
  col1 NUMBER,
  col2 NUMBER,
  CONSTRAINT fk_constraint FOREIGN KEY (col2_fk) REFERENCES
    test1,
  CONSTRAINT ck1 CHECK (col1_pk > 0 and col1 > 0),
  CONSTRAINT ck2 CHECK (col2_fk > 0));
```

An error is returned for the following statements:

ALTER TABLE test1 DROP (col1_pk); —col1_pk is a parent key.

ALTER TABLE test1 DROP (col1); —col1 is referenced by the multicolumn constraint, ck1.

Cascading Constraints

Example:

```
ALTER TABLE emp2
DROP COLUMN employee_id CASCADE CONSTRAINTS;
```

```
table EMP2 altered.
```

```
ALTER TABLE test1
DROP (col1_pk, col2_fk, col1);
```

```
table TEST1 altered.
```


Submitting the following statement drops the EMPLOYEE_ID column, the PRIMARY KEY constraint, and any FOREIGN KEY constraints referencing the PRIMARY KEY constraint for the EMP2 table:

```
ALTER TABLE emp2 DROP COLUMN employee_id CASCADE CONSTRAINTS;
```

If all columns referenced by the constraints defined on the dropped columns are also dropped, CASCADE CONSTRAINTS is not required. For example, assuming that no other referential constraints from other tables refer to the COL1_PK column, it is valid to submit the following statement without the CASCADE CONSTRAINTS clause for the TEST1 table created on the previous page:

```
ALTER TABLE test1 DROP (col1_pk, col2_fk, col1);
```

- Enabling a PRIMARY KEY constraint that was disabled with the CASCADE option does not enable any FOREIGN KEYs that are dependent on the PRIMARY KEY.
- To enable a UNIQUE or PRIMARY KEY constraint, you must have the privileges necessary to create an index on the table.

Renaming Table Columns and Constraints

- Use the RENAME TABLE clause of the ALTER TABLE statement to rename tables.

a

```
ALTER TABLE marketing RENAME to new_marketing;
```

- Use the RENAME COLUMN clause of the ALTER TABLE statement to rename table columns.

b

```
ALTER TABLE new_marketing RENAME COLUMN team_id
TO id;
```

- Use the RENAME CONSTRAINT clause of the ALTER TABLE statement to rename any existing constraint for a table.

c

```
ALTER TABLE new_marketing RENAME CONSTRAINT mktg_pk
TO new_mktg_pk;
```


`RENAME TABLE` clause allows you to rename an existing table in any schema (except the schema *SYS*).To rename a table, you must either be the database owner or the table owner.

When you rename a table column, the new name must not conflict with the name of any existing column in the table. You cannot use any other clauses in conjunction with the `RENAME COLUMN` clause.

The slide examples use the `marketing` table with the `PRIMARY KEY` `mktg_pk` defined on the `id` column.

```
CREATE TABLE marketing (team_id  NUMBER(10),
                        target   VARCHAR2(50),
CONSTRAINT mktg_pk PRIMARY KEY(team_id));
```

Example **a** shows that the `marketing` table is renamed `new_marketing`. Example **b** shows that the `id` column of the `new_marketing` table is renamed `mktg_id` and example **c** shows that `mktg_pk` is renamed `new_mktg_pk`.

When you rename any existing constraint for a table, the new name must not conflict with any of your existing constraint names. You can use the `RENAME CONSTRAINT` clause to rename system-generated constraint names.

Disabling Constraints

- Execute the `DISABLE` clause of the `ALTER TABLE` statement to deactivate an integrity constraint.
- Apply the `CASCADE` option to disable the primary key and it will disable all dependent `FOREIGN KEY` constraints automatically as well.

```
ALTER TABLE emp2
DISABLE CONSTRAINTS emp_dt_pk;
```

```
table EMP2 altered.
```

```
ALTER TABLE dept2
DISABLE primary key CASCADE;
```

```
table DEPT2 altered.
```


You can disable a constraint, without dropping it or re-creating it, by using the `ALTER TABLE` statement with the `DISABLE` clause. You can also disable the primary key or unique key using the `CASCADE` option.

Syntax

```
ALTER    TABLE    table
DISABLE CONSTRAINT constraint [CASCADE];
```

In the syntax:

table	Is the name of the table
constraint	Is the name of the constraint

Guidelines

- You can use the `DISABLE` clause in both the `CREATE TABLE` statement and the `ALTER TABLE` statement.
- The `CASCADE` clause disables dependent integrity constraints.
- Disabling a `UNIQUE` or `PRIMARY KEY` constraint removes the unique index.

Enabling Constraints

- Activate an integrity constraint that is currently disabled in the table definition by using the `ENABLE` clause.

```
ALTER TABLE         emp2
ENABLE CONSTRAINT emp_dt_fk;
```

```
table EMP2 altered.
```

- A `UNIQUE` index is automatically created if you enable a `UNIQUE` key or a `PRIMARY KEY` constraint.

You can enable a constraint without dropping it or re-creating it by using the `ALTER TABLE` statement with the `ENABLE` clause.

Syntax

```
ALTER   TABLE      table
ENABLE  CONSTRAINT constraint;
```

In the syntax:

table	Is the name of the table
constraint	Is the name of the constraint

Guidelines

- If you enable a constraint, that constraint applies to all the data in the table. All the data in the table must comply with the constraint.
- If you enable a `UNIQUE` key or a `PRIMARY KEY` constraint, a `UNIQUE` or `PRIMARY KEY` index is created automatically. If an index already exists, it can be used by these keys.
- You can use the `ENABLE` clause in both the `CREATE TABLE` statement and the `ALTER TABLE` statement.

Constraint States

An integrity constraint defined on a table can be in one of the following states:

- `ENABLE VALIDATE`
- `ENABLE NOVALIDATE`
- `DISABLE VALIDATE`
- `DISABLE NOVALIDATE`

```
ALTER TABLE dept2
ENABLE NOVALIDATE PRIMARY KEY;
```

```
table DEPT2 altered.
```


You can enable or disable integrity constraints at the table level using the `CREATE TABLE` or `ALTER TABLE` statement. You can also set constraints to `VALIDATE` or `NOVALIDATE`, in any combination with `ENABLE` or `DISABLE`, where:

- `ENABLE` ensures that all incoming data conforms to the constraint
- `DISABLE` allows incoming data, regardless of whether it conforms to the constraint
- `VALIDATE` ensures that existing data conforms to the constraint
- `NOVALIDATE` means that some existing data may not conform to the constraint

`ENABLE VALIDATE` is the same as `ENABLE`. The constraint is checked and is guaranteed to hold for all rows. `ENABLE NOVALIDATE` means that the constraint is checked, but it does not have to be true for all rows. This allows existing rows to violate the constraint, while ensuring that all new or modified rows are valid. In an `ALTER TABLE` statement, `ENABLE NOVALIDATE` resumes constraint checking on disabled constraints without first validating all data in the table. `DISABLE NOVALIDATE` is the same as `DISABLE`. The constraint is not checked and is not necessarily true. `DISABLE VALIDATE` disables the constraint, drops the index on the constraint, and disallows any modification of the constrained columns.

Deferring Constraints

Constraints can have the following attributes:

- `DEFERRABLE` or `NOT DEFERRABLE`
- `INITIALLY DEFERRED` or `INITIALLY IMMEDIATE`

```
ALTER TABLE dept2
ADD CONSTRAINT dept2_id_pk
PRIMARY KEY (department_id)
DEFERRABLE INITIALLY DEFERRED;
```

Deferring constraint on creation

```
SET CONSTRAINTS dept2_id_pk IMMEDIATE;
```

Changing a specific constraint attribute

```
ALTER SESSION
SET CONSTRAINTS= IMMEDIATE;
```

Changing all constraints for a session

You can defer checking constraints for validity until the end of the transaction. A constraint is deferred if the system does not check whether the constraint is satisfied, until a `COMMIT` statement is submitted. If a deferred constraint is violated, the database returns an error and the transaction is not committed and it is rolled back. If a constraint is immediate (not deferred), it is checked at the end of each statement. If it is violated, the statement is rolled back immediately. If a constraint causes an action (for example, `DELETE CASCADE`), that action is always taken as part of the statement that caused it, whether the constraint is deferred or immediate. Use the `SET CONSTRAINTS` statement to specify, for a particular transaction, whether a deferrable constraint is checked following each data manipulation language (DML) statement or when the transaction is committed. To create deferrable constraints, you must create a nonunique index for that constraint.

You can define constraints as either deferrable or `NOT DEFERRABLE` (default), and either initially deferred or `INITIALLY IMMEDIATE` (default). These attributes can be different for each constraint.

Usage scenario: Company policy dictates that department number 40 should be changed to 45. Changing the `DEPARTMENT_ID` column affects employees assigned to this department. Therefore, you make the `PRIMARY KEY` and `FOREIGN KEYs` deferrable and initially deferred. You update both department and employee information, and at the time of commit, all the rows are validated.

Difference Between INITIALLY DEFERRED and INITIALLY IMMEDIATE

INITIALLY DEFERRED	Waits until the transaction ends to check the constraint
INITIALLY IMMEDIATE	Checks the constraint at the end of the statement execution

```
CREATE TABLE emp_new_sal (salary NUMBER
        CONSTRAINT sal_ck
        CHECK (salary > 100)
        DEFERRABLE INITIALLY IMMEDIATE,
        bonus NUMBER
        CONSTRAINT bonus_ck
        CHECK (bonus > 0 )
        DEFERRABLE INITIALLY DEFERRED );
```

```
table EMP_NEW_SAL created.
```

ORACLE

A constraint that is defined as deferrable can be specified as either `INITIALLY DEFERRED` or `INITIALLY IMMEDIATE`. The `INITIALLY IMMEDIATE` clause is the default.

In the slide example:

- The `sal_ck` constraint is created as `DEFERRABLE INITIALLY IMMEDIATE`
- The `bonus_ck` constraint is created as `DEFERRABLE INITIALLY DEFERRED`

After creating the `emp_new_sal` table, as shown in the slide, you attempt to insert values into the table and observe the results. When both the `sal_ck` and `bonus_ck` constraints are satisfied, the rows are inserted without an error.

Example 1: Insert a row that violates `sal_ck`. In the `CREATE TABLE` statement, `sal_ck` is specified as an initially immediate constraint. This means that the constraint is verified immediately after the `INSERT` statement and you observe an error.

```
INSERT INTO emp_new_sal VALUES(90,5);
```

```
SQL Error: ORA-02290: check constraint (ORA21.SAL_CK) violated
02290. 00000 -  "check constraint (%s.%s) violated"
```

Example 2: Insert a row that violates `bonus_ck`. In the `CREATE TABLE` statement, `bonus_ck` is specified as deferrable and also initially deferred. Therefore, the constraint is not verified until you `COMMIT` or set the constraint state back to immediate.

```
INSERT INTO emp_new_sal VALUES(110, -1);
```

```
1 rows inserted
```

The row insertion is successful. But you observe an error when you commit the transaction.

```
COMMIT;
```

```
SQL Error: ORA-02091: transaction rolled back
ORA-02290: check constraint (ORA21.BONUS_CK) violated
02091. 00000 -  "transaction rolled back"
```

The commit failed due to constraint violation. Therefore, at this point, the transaction is rolled back by the database.

Example 3: Set the `DEFERRED` status to all constraints that can be deferred. Note that you can also set the `DEFERRED` status to a single constraint if required.

```
SET CONSTRAINTS ALL DEFERRED;
```

```
constraints ALL succeeded.
```

Now, if you attempt to insert a row that violates the `sal_ck` constraint, the statement is executed successfully.

```
INSERT INTO emp_new_sal VALUES(90,5);
```

```
1 rows inserted
```

However, you observe an error when you commit the transaction. The transaction fails and is rolled back. This is because both the constraints are checked upon `COMMIT`.

```
COMMIT;
```

```
SQL Error: ORA-02091: transaction rolled back
ORA-02290: check constraint (ORA21.SAL_CK) violated
02091. 00000 -  "transaction rolled back"
```

Example 4: Set the `IMMEDIATE` status to both the constraints that were set as `DEFERRED` in the previous example.

```
SET CONSTRAINTS ALL IMMEDIATE;
```

```
constraints ALL succeeded.
```

You observe an error if you attempt to insert a row that violates either `sal_ck` or `bonus_ck`.

```
INSERT INTO emp_new_sal VALUES(110, -1);
```

```
SQL Error: ORA-02290: check constraint (ORA21.BONUS_CK) violated
02290. 00000 -  "check constraint (%s.%s) violated"
```

Note: If you create a table without specifying constraint deferability, the constraint is checked immediately at the end of each statement. For example, with the `CREATE TABLE` statement of the `newemp_details` table, if you do not specify the `newemp_det_pk` constraint deferability, the constraint is checked immediately.

```
CREATE TABLE newemp_details(emp_id NUMBER, emp_name
VARCHAR2(20),
CONSTRAINT newemp_det_pk PRIMARY KEY(emp_id));
```

When you attempt to defer the `newemp_det_pk` constraint that is not deferrable, you observe the following error:

```
SET CONSTRAINT newemp_det_pk DEFERRED;
```

```
SQL Error: ORA-02447: cannot defer a constraint that is not deferrable
```

DROP TABLE … PURGE

```
DROP TABLE emp_new_sal PURGE;
```

```
table DEPT80 dropped.
```


Oracle Database provides a feature for dropping tables. When you drop a table, the database does not immediately release the space associated with the table. Rather, the database renames the table and places it in a recycle bin, where it can later be recovered with the FLASHBACK TABLE statement if you find that you dropped the table in error. If you want to immediately release the space associated with the table at the time you issue the DROP TABLE statement, include the PURGE clause as shown in the statement in the slide.

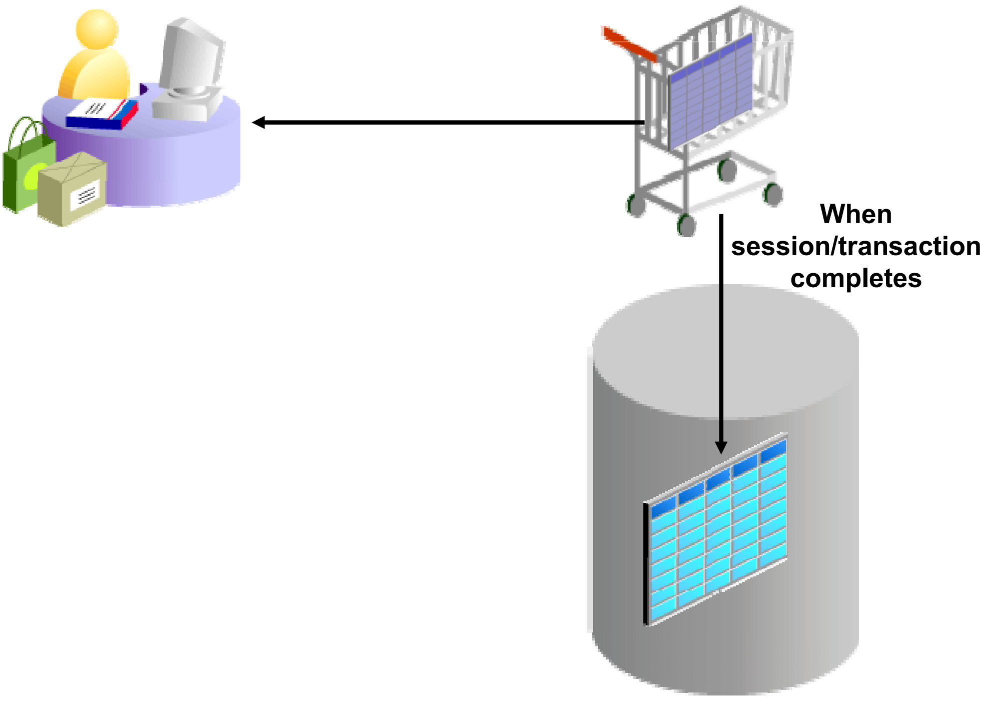
Specify PURGE only if you want to drop the table and release the space associated with it in a single step. If you specify PURGE, the database does not place the table and its dependent objects into the recycle bin.

Using this clause is equivalent to first dropping the table and then purging it from the recycle bin. This clause saves you one step in the process. It also provides enhanced security if you want to prevent sensitive material from appearing in the recycle bin.

Lesson Agenda

- Managing constraints:
  - Adding and dropping a constraint
  - Enabling and disabling a constraint
  - Deferring constraints
- Creating and using temporary tables
- Creating and using external tables

Temporary Tables



A temporary table is a table that holds data that exists only for the duration of a transaction or session. Data in a temporary table is private to the session, which means that each session can see and modify only its own data.

Temporary tables are useful in applications where a result set must be buffered. For example, a shopping cart in an online application can be a temporary table. Each item is represented by a row in the temporary table. While you are shopping in an online store, you can keep on adding or removing items from your cart. During the session, this cart data is private. After you finalize your shopping and make the payments, the application moves the row for the chosen cart to a permanent table. At the end of the session, the data in the temporary table is automatically dropped.

Because temporary tables are statically defined, you can create indexes for them. Indexes created on temporary tables are also temporary. The data in the index has the same session or transaction scope as the data in the temporary table. You can also create a view or trigger on a temporary table.

Creating a Temporary Table

```
CREATE GLOBAL TEMPORARY TABLE cart(n NUMBER,d DATE)
ON COMMIT DELETE ROWS;
```

1

```
CREATE GLOBAL TEMPORARY TABLE today_sales
ON COMMIT PRESERVE ROWS AS
     SELECT * FROM orders
              WHERE order_date = SYSDATE;
```

2

To create a temporary table, you can use the following command:

```
CREATE GLOBAL TEMPORARY TABLE tablename
ON COMMIT [PRESERVE | DELETE] ROWS
```

By associating one of the following settings with the `ON COMMIT` clause, you can decide whether the data in the temporary table is transaction-specific (default) or session-specific.

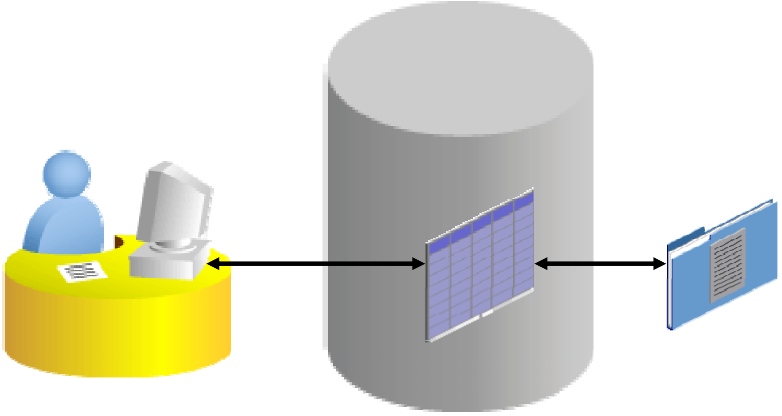
1. `DELETE ROWS`: As shown in example 1 in the slide, the `DELETE ROWS` setting creates a temporary table that is transaction-specific. A session becomes bound to the temporary table with a transaction's first insert into the table. The binding goes away at the end of the transaction. The database truncates the table (delete all rows) after each commit.
2. `PRESERVE ROWS`: As shown in example 2 in the slide, the `PRESERVE ROWS` setting creates a temporary table that is session-specific. Each sales representative session can store its own sales data for the day in the table. When a salesperson performs first insert on the `today_sales` table, his or her session gets bound to the `today_sales` table. This binding goes away at the end of the session or by issuing a `TRUNCATE` of the table in the session. The database truncates the table when you terminate the session.

When you create a temporary table in an Oracle database, you create a static table definition. Like permanent tables, temporary tables are defined in the data dictionary. However, temporary tables and their indexes do not automatically allocate a segment when created. Instead, temporary segments are allocated when data is first inserted. Until data is loaded in a session, the table appears empty.

Lesson Agenda

- Managing constraints:
  - Adding and dropping a constraint
  - Enabling and disabling a constraint
  - Deferring constraints
- Creating and using temporary tables
- Creating and using external tables

External Tables



An external table is a read-only table whose metadata is stored in the database but whose data is stored outside the database. This external table definition can be thought of as a view that is used for running any SQL query against external data without requiring that the external data first be loaded into the database. The external table data can be queried and joined directly and in parallel without requiring that the external data first be loaded in the database. You can use SQL, PL/SQL, and Java to query the data in an external table.

The main difference between external tables and regular tables is that externally organized tables are read-only. No data manipulation language (DML) operations are possible, and no indexes can be created on them. However, you can create an external table, and thus unload data, by using the `CREATE TABLE AS SELECT` command.

The Oracle Server provides two major access drivers for external tables. One, the loader access driver (or `ORACLE_LOADER`) is used for reading data from external files whose format can be interpreted by the SQL*Loader utility. Note that not all SQL*Loader functionality is supported with external tables. The `ORACLE_DATAPUMP` access driver can be used to both import and export data by using a platform-independent format. The `ORACLE_DATAPUMP` access driver writes rows from a `SELECT` statement to be loaded into an external table as part of a `CREATE TABLE ...ORGANIZATION EXTERNAL...AS SELECT` statement. You can then use `SELECT` to read data out of that data file. You can also create an external table definition on another system and use that data file. This allows data to be moved between Oracle databases.

Creating a Directory for the External Table

Create a `DIRECTORY` object that corresponds to the directory on the file system where the external data source resides.

```
CREATE OR REPLACE DIRECTORY emp_dir
AS '/…/emp_dir';

GRANT READ ON DIRECTORY emp_dir TO ora_21;
```


Use the `CREATE DIRECTORY` statement to create a directory object. A directory object specifies an alias for a directory on the server's file system where an external data source resides. You can use directory names when referring to an external data source, rather than hard code the operating system path name, for greater file management flexibility.

You must have `CREATE ANY DIRECTORY` system privileges to create directories. When you create a directory, you are automatically granted the `READ` and `WRITE` object privileges and can grant `READ` and `WRITE` privileges to other users and roles. The DBA can also grant these privileges to other users and roles.

A user needs `READ` privileges for all directories used in external tables to be accessed and `WRITE` privileges for the log, bad, and discard file locations being used.

In addition, a `WRITE` privilege is necessary when the external table framework is being used to unload data.

Oracle also provides the `ORACLE_DATAPUMP` type, with which you can unload data (that is, read data from a table in the database and insert it into an external table) and then reload it into an Oracle database. This is a one-time operation that can be done when the table is created. After the creation and initial population is done, you cannot update, insert, or delete any rows.

Syntax

```
CREATE [OR REPLACE] DIRECTORY AS 'path_name';
```

In the syntax:

`OR REPLACE`	Specify `OR REPLACE` to re-create the directory database object if it already exists. You can use this clause to change the definition of an existing directory without dropping, re-creating, and regranting database object privileges previously granted on the directory. Users who were previously granted privileges on a redefined directory can continue to access the directory without requiring that the privileges be regranted.
`directory`	Specify the name of the directory object to be created. The maximum length of the directory name is 30 bytes. You cannot qualify a directory object with a schema name.
`'path_name'`	Specify the full path name of the operating system directory to be accessed. The path name is case-sensitive.

Creating an External Table

```
CREATE TABLE <table_name>
   ( <col_name> <datatype>, … )
ORGANIZATION EXTERNAL
    (TYPE <access_driver_type>
     DEFAULT DIRECTORY <directory_name>
     ACCESS PARAMETERS
      (… ) )
      LOCATION ('<location_specifier>')
REJECT LIMIT [0 | <number> | UNLIMITED];
```


You create external tables by using the `ORGANIZATION EXTERNAL` clause of the `CREATE TABLE` statement. You are not, in fact, creating a table. Rather, you are creating metadata in the data dictionary that you can use to access external data. You use the `ORGANIZATION` clause to specify the order in which the data rows of the table are stored. By specifying `EXTERNAL` in the `ORGANIZATION` clause, you indicate that the table is a read-only table located outside the database. Note that the external files must already exist outside the database.

`TYPE <access_driver_type>` indicates the access driver of the external table. The access driver is the application programming interface (API) that interprets the external data for the database. If you do not specify `TYPE`, Oracle uses the default access driver, `ORACLE_LOADER`. The other option is `ORACLE_DATAPUMP`.

You use the `DEFAULT DIRECTORY` clause to specify one or more Oracle database directory objects that correspond to directories on the file system where the external data sources may reside.

The optional `ACCESS PARAMETERS` clause enables you to assign values to the parameters of the specific access driver for this external table.

Use the `LOCATION` clause to specify one external locator for each external data source. Usually, *<location_specifier>* is a file, but it need not be.

The `REJECT LIMIT` clause enables you to specify how many conversion errors can occur during a query of the external data before an Oracle error is returned and the query is aborted. The default value is `0`.

The syntax for using the `ORACLE_DATAPUMP` access driver is as follows:

```
CREATE TABLE extract_emps
ORGANIZATION EXTERNAL (TYPE ORACLE_DATAPUMP
                       DEFAULT DIRECTORY …
                       ACCESS PARAMETERS (… )
                       LOCATION (…)
                       PARALLEL 4
                       REJECT LIMIT UNLIMITED
AS
SELECT * FROM …;
```

Creating an External Table by Using ORACLE_LOADER

```
CREATE TABLE oldemp (fname char(25), lname
CHAR(25))
ORGANIZATION EXTERNAL
(TYPE ORACLE_LOADER
 DEFAULT DIRECTORY emp_dir
 ACCESS PARAMETERS
 (RECORDS DELIMITED BY NEWLINE
  FIELDS(fname POSITION ( 1:20) CHAR,
  lname POSITION (22:41) CHAR))
 LOCATION ('emp.dat'));
```

```
table OLDEMP created.
```


Assume that there is a flat file that has records in the following format:

```
10,jones,11-Dec-1934
20,smith,12-Jun-1972
```

Records are delimited by new lines. The name of the file is `/emp_dir/emp.dat`.

To convert this file as the data source for an external table, whose metadata will reside in the database, you must perform the following steps:

1. Create a directory object, `emp_dir`, as follows:
   `CREATE DIRECTORY emp_dir AS '/emp_dir' ;`
2. Run the `CREATE TABLE` command shown in the slide.

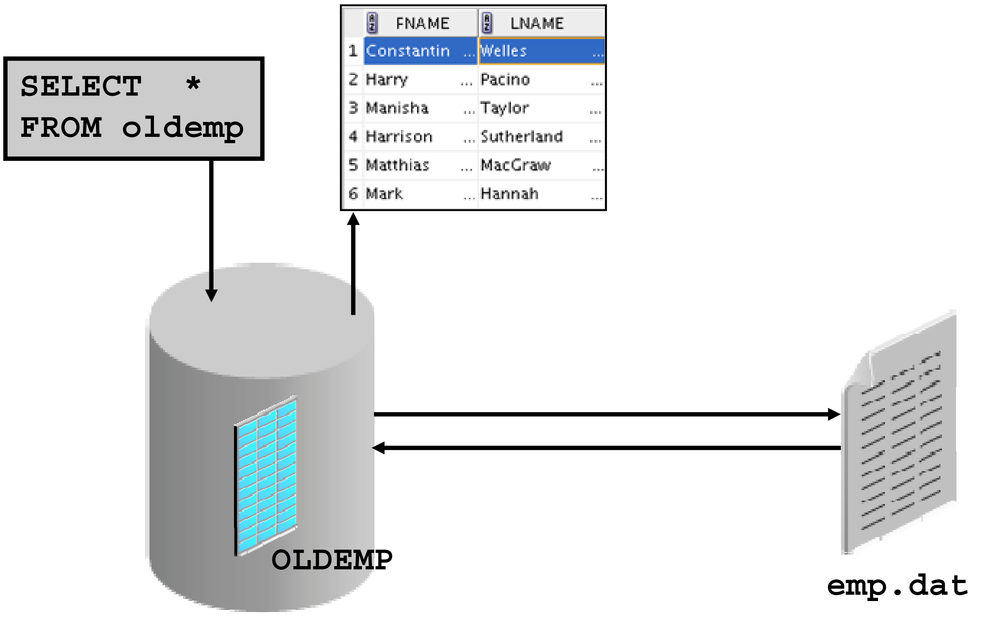
The example in the slide illustrates the table specification to create an external table for the file:

```
/emp_dir/emp.dat
```

In the example, the TYPE specification is given only to illustrate its use. ORACLE_LOADER is the default access driver if not specified. The ACCESS PARAMETERS option provides values to parameters of the specific access driver, which are interpreted by the access driver, not by the Oracle Server.

After the CREATE TABLE command executes successfully, the OLDEMP external table can be described and queried in the same way as a relational table.

Querying External Tables



An external table does not describe any data that is stored in the database. It does not describe how data is stored in the external source. Instead, it describes how the external table layer must present the data to the server. It is the responsibility of the access driver and the external table layer to do the necessary transformations required on the data in the data file so that it matches the external table definition.

When the database server accesses data in an external source, it calls the appropriate access driver to get the data from an external source in a form that the database server expects.

It is important to remember that the description of the data in the data source is separate from the definition of the external table. The source file can contain more or fewer fields than there are columns in the table. Also, the data types for fields in the data source can be different from the columns in the table. The access driver takes care of ensuring that the data from the data source is processed so that it matches the definition of the external table.

Creating an External Table by Using ORACLE_DATAPUMP: Example

```
CREATE TABLE emp_ext
  (employee_id, first_name, last_name)
   ORGANIZATION EXTERNAL
     (
      TYPE ORACLE_DATAPUMP
      DEFAULT DIRECTORY emp_dir
      LOCATION
        ('emp1.exp','emp2.exp')
     )
   PARALLEL
AS
SELECT employee_id, first_name, last_name
FROM   employees;
```

```
table EMP_EXT created.
```


You can perform the unload and reload operations with external tables by using the ORACLE_DATAPUMP access driver.

Note: In the context of external tables, loading data refers to the act of data being read from an external table and loaded into a table in the database. Unloading data refers to the act of reading data from a table and inserting it into an external table.

The example in the slide illustrates the table specification to create an external table by using the ORACLE_DATAPUMP access driver. Data is then populated into the two files: emp1.exp and emp2.exp.

To populate data read from the EMPLOYEES table into an external table, you must perform the following steps:

1. Create a directory object, emp_dir, as follows:
   `CREATE DIRECTORY emp_dir AS '/emp_dir' ;`
2. Run the CREATE TABLE command shown in the slide.

Note: The emp_dir directory is the same as created in the previous example of using ORACLE_LOADER.

You can query the external table by executing the following code:

```
SELECT * FROM emp_ext;
```

Quiz

A `FOREIGN KEY` constraint enforces the following action: When the data in the parent key is deleted, all the rows in the child table that depend on the deleted parent key values are also deleted.

a. True

b. False

ORACLE

Answer: b

Summary

In this lesson, you should have learned how to:

- Manage constraints
- Create and use temporary tables
- Create and use external tables

ORACLE

In this lesson, you learned how to perform the following tasks for schema object management:

- Alter tables to add or modify columns or constraints.
- Use the `ORGANIZATION EXTERNAL` clause of the `CREATE TABLE` statement to create an external table. An external table is a read-only table whose metadata is stored in the database but whose data is stored outside the database.
- Use external tables to query data without first loading it into the database.

Practice 5: Overview

This practice covers the following topics:

- Adding and dropping constraints
- Deferring constraints
- Creating external tables

In this practice, you use the `ALTER TABLE` command to add, drop and defer constraints. You create external tables.

Retrieving Data by Using Subqueries

Objectives

After completing this lesson, you should be able to:

- Write a multiple-column subquery
- Use scalar subqueries in SQL
- Solve problems with correlated subqueries
- Use the `EXISTS` and `NOT EXISTS` operators
- Use the `WITH` clause

In this lesson, you learn how to write multiple-column subqueries and subqueries in the `FROM` clause of a `SELECT` statement. You also learn how to solve problems by using scalar, correlated subqueries and the `WITH` clause.

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

Retrieving Data by Using a Subquery as a Source

```
SELECT department_name, city
FROM   departments
NATURAL JOIN (SELECT l.location_id, l.city, l.country_id
              FROM   locations l
              JOIN   countries c
              ON(l.country_id = c.country_id)
              JOIN regions
              USING(region_id)
              WHERE region_name = 'Europe');
```

	DEPARTMENT_NAME	CITY
1	Human Resources	London
2	Sales	Oxford
3	Public Relations	Munich

ORACLE

You can use a subquery in the FROM clause of a SELECT statement, which is very similar to how views are used. A subquery in the FROM clause of a SELECT statement is also called an *inline* view. A subquery in the FROM clause of a SELECT statement defines a data source for that particular SELECT statement, and only that SELECT statement. As with a database view, the SELECT statement in the subquery can be as simple or as complex as you like.

When a database view is created, the associated SELECT statement is stored in the data dictionary. In situations where you do not have the necessary privileges to create database views, or when you would like to test the suitability of a SELECT statement to become a view, you can use an inline view.

With inline views, you can have all the code needed to support the query in one place. This means that you can avoid the complexity of creating a separate database view. The example in the slide shows how to use an inline view to display the department name and the city in Europe. The subquery in the FROM clause fetches the location ID, city name, and the country by joining three different tables. The output of the inner query is considered as a table for the outer query. The inner query is similar to that of a database view but does not have any physical name.

You can display the same output as in the example in the slide by performing the following two steps:

1. Create a database view:

   ```
   CREATE OR REPLACE VIEW european_cities
   AS
   SELECT l.location_id, l.city, l.country_id
   FROM   locations l
   JOIN   countries c
   ON(l.country_id = c.country_id)
   JOIN regions USING(region_id)
   WHERE region_name = 'Europe';
   ```

2. Join the `EUROPEAN_CITIES` view with the `DEPARTMENTS` table:

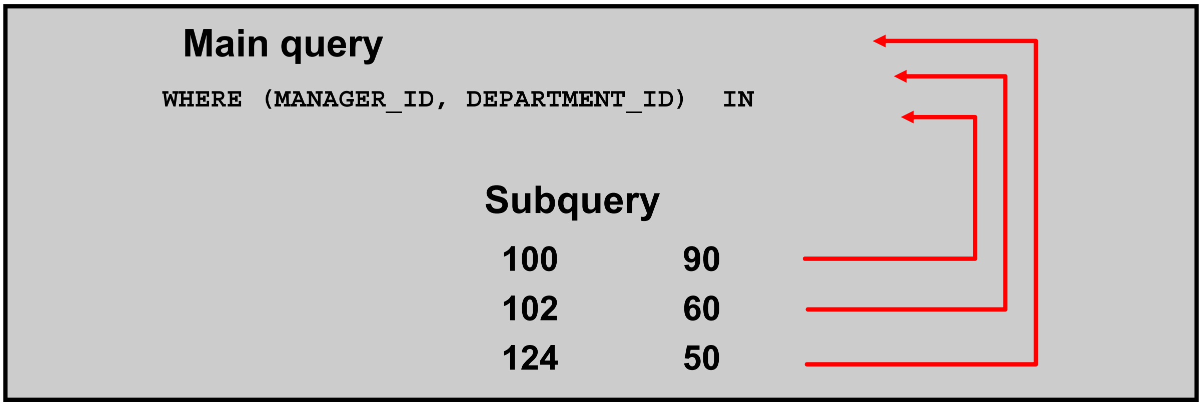
   ```
   SELECT department_name, city
   FROM   departments
   NATURAL JOIN european_cities;
   ```

Note: You learned how to create database views in the lesson titled *Creating Views.*

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

Multiple-Column Subqueries



Each row of the main query is compared to values from a multiple-row and multiple-column subquery.

So far, you have written single-row subqueries and multiple-row subqueries where only one column is returned by the inner `SELECT` statement and this is used to evaluate the expression in the parent `SELECT` statement. If you want to compare two or more columns, you must write a compound `WHERE` clause using logical operators. Using multiple-column subqueries, you can combine duplicate `WHERE` conditions into a single `WHERE` clause.

Syntax

```
SELECT      column, column, ...
FROM  table
WHERE (column, column, ...) IN
              (SELECT column, column, ...
                      FROM   table
                      WHERE  condition);
```

The graphic in the slide illustrates that the values of `MANAGER_ID` and `DEPARTMENT_ID` from the main query are being compared with the `MANAGER_ID` and `DEPARTMENT_ID` values retrieved by the subquery. Because the number of columns that are being compared is more than one, the example qualifies as a multiple-column subquery.

Note: Before you run the examples in the next few slides, you need to create the `empl_demo` table and populate data into it by using the `lab_06_insert_empdata.sql` file.

Column Comparisons

Multiple-column comparisons involving subqueries can be:

- Pairwise comparisons
- Nonpairwise comparisons

Pairwise Versus Nonpairwise Comparisons

Multiple-column comparisons involving subqueries can be nonpairwise comparisons or pairwise comparisons. If you consider the example “Display the details of the employees who work in the same department, and have the same manager, as ‘Daniel’?,” you get the correct result with the following statement:

```
SELECT first_name, last_name, manager_id, department_id
FROM empl_demo
WHERE manager_id IN (SELECT manager_id
                     FROM empl_demo
                     WHERE first_name = 'Daniel')
AND department_id IN (SELECT department_id
                      FROM empl_demo
                      WHERE first_name = 'Daniel');
```

There is only one “Daniel” in the `EMPL_DEMO` table (Daniel Faviet, who is managed by employee 108 and works in department 100). However, if the subqueries return more than one row, the result might not be correct. For example, if you run the same query but substitute “John” for “Daniel,” you get an incorrect result. This is because the combination of `department_id` and `manager_id` is important. To get the correct result for this query, you need a pairwise comparison.

Pairwise Comparison Subquery

Display the details of the employees who are managed by the same manager and work in the same department as the employees with EMPLOYEE_ID 199 or 174.

```
SELECT employee_id, manager_id, department_id
FROM employees
WHERE (manager_id, department_id) IN
                            (SELECT manager_id, department_id
                             FROM employees
                             WHERE employee_id IN (174, 199))
AND employee_id NOT IN (174,199);
```

	EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
1	141	124	50
2	142	124	50
3	143	124	50
4	144	124	50

…

The example shows a pairwise comparison of the columns. It compares the values in the MANAGER_ID column and the DEPARTMENT_ID column of each row in the EMPLOYEES table with the values in the MANAGER_ID column and the DEPARTMENT_ID column for the employees with the EMPLOYEE_ID 199 or 174.

First, the subquery to retrieve the MANAGER_ID and DEPARTMENT_ID values for the employees with the EMPLOYEE_ID 199 or 174 is executed. These values are compared with the MANAGER_ID column and the DEPARTMENT_ID column of each row in the EMPLOYEES table. If the values match, the row is displayed. In the output, the records of the employees with the EMPLOYEE_ID 199 or 174 will not be displayed. The output of the query in the slide follows.

Nonpairwise Comparison Subquery

Display the details of the employees who are managed by the same manager as the employees with EMPLOYEE_ID 174 or 141 and work in the same department as the employees with EMPLOYEE_ID 174 or 141.

```
SELECT employee_id, manager_id, department_id
FROM employees
WHERE manager_id IN
                  (SELECT manager_id
                   FROM employees
                   WHERE employee_id IN (174,141))
AND department_id IN
                  (SELECT department_id
                   FROM employees
                   WHERE employee_id IN (174,141))
AND employee_id NOT IN(174,141);
```

	EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
1	142	124	50
2	143	124	50

...

The example shows a nonpairwise comparison of the columns. It displays the EMPLOYEE_ID, MANAGER_ID, and DEPARTMENT_ID of any employee whose manager ID matches any of the manager IDs of employees whose employee IDs are either 174 or 141 and DEPARTMENT_ID match any of the department IDs of employees whose employee IDs are either 174 or 141.

First, the subquery to retrieve the MANAGER_ID values for the employees with the EMPLOYEE_ID 174 or 141 is executed. Similarly, the second subquery to retrieve the DEPARTMENT_ID values for the employees with the EMPLOYEE_ID 174 or 141 is executed. The retrieved values of the MANAGER_ID and DEPARTMENT_ID columns are compared with the MANAGER_ID and DEPARTMENT_ID column for each row in the EMPLOYEES table. If the MANAGER_ID column of the row in the EMPLOYEES table matches with any of the values of the MANAGER_ID retrieved by the inner subquery and if the DEPARTMENT_ID column of the row in the EMPLOYEES table matches with any of the values of the DEPARTMENT_ID retrieved by the second subquery, the record is displayed.

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

Scalar Subquery Expressions

- A scalar subquery expression is a subquery that returns exactly one column value from one row.
- Scalar subqueries can be used in:
  - The condition and expression part of `DECODE` and `CASE`
  - All clauses of `SELECT` except `GROUP BY`
  - The `SET` clause and `WHERE` clause of an `UPDATE` statement

A subquery that returns exactly one column value from one row is also referred to as a scalar subquery. Multiple-column subqueries that are written to compare two or more columns, using a compound `WHERE` clause and logical operators, do not qualify as scalar subqueries.

The value of the scalar subquery expression is the value of the select list item of the subquery. If the subquery returns 0 rows, the value of the scalar subquery expression is `NULL`. If the subquery returns more than one row, the Oracle Server returns an error. The Oracle Server has always supported the usage of a scalar subquery in a `SELECT` statement. You can use scalar subqueries in:

- The condition and expression part of `DECODE` and `CASE`
- All clauses of `SELECT` except `GROUP BY`
- The `SET` clause and `WHERE` clause of an `UPDATE` statement

However, scalar subqueries are not valid expressions in the following places:

- In the `RETURNING` clause of data manipulation language (DML) statements
- As the basis of a function-based index
- In `GROUP BY` clauses, `CHECK` constraints
- In `CONNECT BY` clauses
- In statements that are unrelated to queries, such as `CREATE PROFILE`

Scalar Subqueries: Examples

- Scalar subqueries in `CASE` expressions:

```
SELECT employee_id, last_name,
       (CASE
        WHEN department_id =                    20
            (SELECT department_id
             FROM departments
             WHERE location_id = 1800)
         THEN 'Canada' ELSE 'USA' END) location
FROM   employees;
```

- Scalar subqueries in the `SELECT` statement:

```
select department_id, department_name,
      (select count(*)
       from   employees e
       where  e.department_id = d.department_id) as emp_count
from   departments d;
```

ORACLE

The first example in the slide demonstrates that scalar subqueries can be used in `CASE` expressions. The inner query returns the value 20, which is the department `ID` of the department whose location `ID` is 1800. The `CASE` expression in the outer query uses the result of the inner query to display the employee `ID`, last names, and a value of Canada or USA, depending on whether the department `ID` of the record retrieved by the outer query is 20.

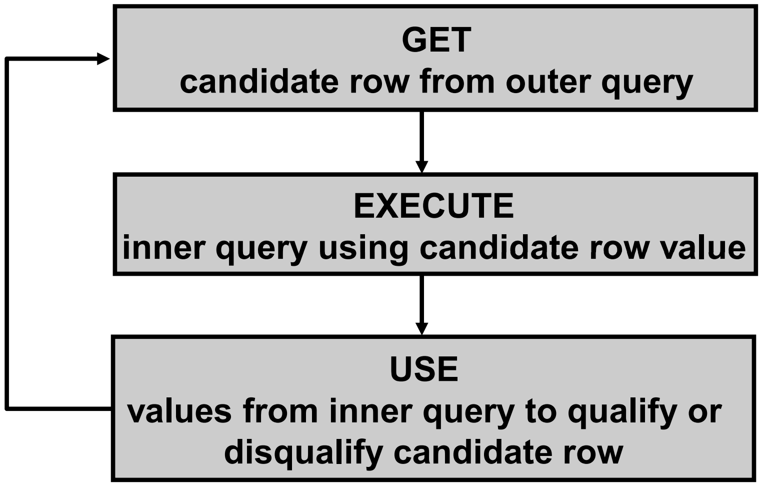
The second example in the slide demonstrates that scalar subqueries can be used in `SELECT` statements 13

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

Correlated Subqueries

Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.



The Oracle Server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a `SELECT`, `UPDATE`, or `DELETE` statement.

Nested Subqueries Versus Correlated Subqueries

With a normal nested subquery, the inner `SELECT` query runs first and executes once, returning values to be used by the main query. A correlated subquery, however, executes once for each candidate row considered by the outer query. That is, the inner query is driven by the outer query.

Nested Subquery Execution

- The inner query executes first and finds a value.
- The outer query executes once, using the value from the inner query.

Correlated Subquery Execution

- Get a candidate row (fetched by the outer query).
- Execute the inner query by using the value of the candidate row.
- Use the values resulting from the inner query to qualify or disqualify the candidate.
- Repeat until no candidate row remains.

Correlated Subqueries

The subquery references a column from a table in the parent query.

```
SELECT column1, column2, ...
FROM   table1   Outer_table
WHERE  column1 operator
                        (SELECT  column1, column2
                         FROM    table2
                         WHERE   expr1 =
                                Outer_table.expr2);
```

ORACLE

A correlated subquery is one way of reading every row in a table and comparing values in each row against related data. It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query. That is, you use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.

The Oracle Server performs a correlated subquery when the subquery references a column from a table in the parent query.

Note: You can use the `ANY` and `ALL` operators in a correlated subquery.

Using Correlated Subqueries: Example 1

Find all employees who earn more than the average salary in their department.

```
SELECT last_name, salary, department_id
FROM   employees outer_table
WHERE  salary >
                (SELECT AVG(salary)
                 FROM   employees inner_table
                 WHERE inner_table.department_id =
                 outer_table.department_id);
```

	LAST_NAME	SALARY	DEPARTMENT_ID
1	King	24000	90
2	Hunold	9000	60
3	Ernst	6000	60
4	Greenberg	12008	100
5	Faviet	9000	100
6	Raphaely	11000	30

....

Each time a row from the outer query is processed, the inner query is evaluated.

The example in the slide finds which employees earn more than the average salary of their department. In this case, the correlated subquery specifically computes the average salary for each department.

Because both the outer query and inner query use the `EMPLOYEES` table in the `FROM` clause, an alias is given to `EMPLOYEES` in the outer `SELECT` statement for clarity. The alias makes the entire `SELECT` statement more readable. Without the alias, the query would not work properly because the inner statement would not be able to distinguish the inner table column from the outer table column.

The correlated subquery performs the following steps for each row of the `EMPLOYEES` table:

1. The `department_id` of the row is determined.
2. The `department_id` is then used to evaluate the parent query.
3. If the salary in that row is greater than the average salary of the departments of that row, then the row is returned.

The subquery is evaluated once for each row of the `EMPLOYEES` table.

Using Correlated Subqueries: Example 2

Display details of highest earning employee in each department.

```
SELECT department_id, employee_id, salary
FROM EMPLOYEES e
WHERE 1 =
         (SELECT  COUNT(DISTINCT salary)
          FROM EMPLOYEES
          WHERE e.department_id = department_id
          AND e.salary <= salary)
```

	DEPARTMENT_ID	EMPLOYEE_ID	SALARY
1	90	100	24000
2	60	103	9000
3	100	108	12008
4	30	114	11000

….

ORACLE

The example in the slide displays the details of highest earning employees in each department. The Oracle Server evaluates a correlated subquery as follows:

1. Select a row from the table specified in the outer query. This will be the current candidate row.
2. Store the value of the column referenced in the subquery from this candidate row. (In the example in the slide, the column referenced in the subquery is `e.salary`.)
3. Perform the subquery with its condition referencing the value from the outer query's candidate row. (In the example in the slide, the `COUNT(DISTINCT salary)` group function is evaluated based on the value of the `E.SALARY` column obtained in step 2.)
4. Evaluate the `WHERE` clause of the outer query on the basis of results of the subquery performed in step 3. This determines whether the candidate row is selected for output. (In the example, the number of times an employee has changed jobs, evaluated by the subquery, is compared with 2 in the `WHERE` clause of the outer query. If the condition is satisfied, that employee record is displayed.)
5. Repeat the procedure for the next candidate row of the table, and so on, until all the rows in the table have been processed.

The correlation is established by using an element from the outer query in the subquery. In this example, you compare `EMPLOYEE_ID` from the table in the subquery with `EMPLOYEE_ID` from the table in the outer query.

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

Using the EXISTS Operator

- The EXISTS operator tests for existence of rows in the results set of the subquery.
- If a subquery row value is found:
  - The search does not continue in the inner query
  - The condition is flagged TRUE
- If a subquery row value is not found:
  - The condition is flagged FALSE
  - The search continues in the inner query

With nesting SELECT statements, all logical operators are valid. In addition, you can use the EXISTS operator. This operator is frequently used with correlated subqueries to test whether a value retrieved by the outer query exists in the results set of the values retrieved by the inner query. If the subquery returns at least one row, the operator returns TRUE. If the value does not exist, it returns FALSE. Accordingly, NOT EXISTS tests whether a value retrieved by the outer query is not a part of the results set of the values retrieved by the inner query.

Using the EXISTS Operator

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees outer
WHERE  EXISTS ( SELECT NULL
                FROM   employees
                WHERE  manager_id =
                       outer.employee_id);
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	100	King	AD_PRES	90
2	101	Kochhar	AD_VP	90
3	102	De Haan	AD_VP	90
4	103	Hunold	IT_PROG	60
5	108	Greenberg	FI_MGR	100
6	114	Raphaely	PU_MAN	30
7	120	Weiss	ST_MAN	50
8	121	Fripp	ST_MAN	50

...

The EXISTS operator ensures that the search in the inner query does not continue when at least one match is found for the manager and employee number by the condition:

```
WHERE manager_id = outer.employee_id
```

Note that the inner SELECT query does not need to return a specific value, so a constant can be selected.

Find All Departments That Do Not Have Any Employees

```
SELECT department_id, department_name
FROM departments d
WHERE NOT EXISTS (SELECT NULL
                  FROM   employees
                  WHERE  department_id = d.department_id);
```

	DEPARTMENT_ID	DEPARTMENT_NAME
1	120	Treasury
2	130	Corporate Tax
3	140	Control And Credit
4	150	Shareholder Services
5	160	Benefits
6	170	Manufacturing
7	180	Construction
8	190	Contracting
9	200	Operations
10	210	IT Support

…

All Rows Fetched: 16

Using the NOT EXISTS Operator

Alternative Solution

A NOT IN construct can be used as an alternative for a NOT EXISTS operator, as shown in the following example:

```
SELECT department_id, department_name
FROM   departments
WHERE  department_id NOT IN (SELECT department_id
                             FROM   employees);
```

However, NOT IN evaluates to FALSE if any member of the set is a NULL value. Therefore, your query will not return any rows even if there are rows in the departments table that satisfy the WHERE condition.

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

`WITH` Clause

- Using the `WITH` clause, you can use the same query block in a `SELECT` statement when it occurs more than once within a complex query.
- The `WITH` clause retrieves the results of a query block and stores it in the user's temporary tablespace.
- The `WITH` clause may improve performance.

Using the `WITH` clause, you can define a query block before using it in a query. The `WITH` clause (formally known as `subquery_factoring_clause`) enables you to reuse the same query block in a `SELECT` statement when it occurs more than once within a complex query. This is particularly useful when a query has many references to the same query block and there are joins and aggregations.

Using the `WITH` clause, you can reuse the same query when it is costly to evaluate the query block and it occurs more than once within a complex query. Using the `WITH` clause, the Oracle Server retrieves the results of a query block and stores it in the user's temporary tablespace. This can improve performance.

`WITH` Clause Benefits

- Makes the query easy to read
- Evaluates a clause only once, even if it appears multiple times in the query
- In most cases, may improve performance for large queries

WITH Clause: Example

```
WITH CNT_DEPT AS
(
SELECT department_id,
 COUNT(1) NUM_EMP
FROM EMPLOYEES
GROUP BY department_id
)
SELECT employee_id,
 SALARY/NUM_EMP
FROM EMPLOYEES E
JOIN CNT_DEPT C
ON (e.department_id = c.department_id);
```

	EMPLOYEE_ID	SALARY/NUM_EMP
1	100	8000
2	103	1800
3	108	2001.333333333333333333333333333333333333
4	110	1366.666666666666666666666666666666666667
5	114	1833.333333333333333333333333333333333333
6	137	80
7	139	60

...

ORACLE

The SQL code in the slide is an example of a situation in which you can improve performance and write SQL more simply by using the `WITH` clause. The query creates the query name as `CNT_DEPT` then uses them in the body of the main query. Here, you perform a math operation by dividing the salary of employee with the total number of employees in each department. Internally, the `WITH` clause is resolved either as an inline view or a temporary table. The optimizer chooses the appropriate resolution depending on the cost or benefit of temporarily storing the results of the `WITH` clause.

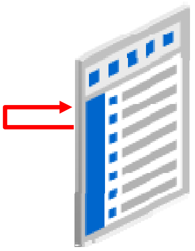
WITH Clause Usage Notes

- It is used only with `SELECT` statements.
- A query name is visible to all `WITH` element query blocks (including their subquery blocks) defined after it and the main query block itself (including its subquery blocks).
- When the query name is the same as an existing table name, the parser searches from the inside out, and the query block name takes precedence over the table name.
- The `WITH` clause can hold more than one query. Each query is then separated by a comma.

Recursive WITH Clause

The Recursive WITH clause:

- Enables formulation of recursive queries
- Creates a query with a name, called the Recursive WITH element name
- Contains two types of query block members: an anchor and a recursive
- Is ANSI-compatible



The WITH clause has been extended to enable formulation of recursive queries.

Recursive WITH defines a recursive query with a name, the Recursive WITH element name. The Recursive WITH element definition must contain at least two query blocks: an anchor member and a recursive member. There can be multiple anchor members, but there can be only a single recursive member. The anchor member must appear before the recursive member, and it cannot reference *query_name*. The anchor member can be composed of one or more query blocks combined by the set operators for example , UNION ALL, UNION, INTERSECT or MINUS. The recursive member must follow the anchor member and must reference *query_name* exactly once. You must combine the recursive member with the anchor member using the UNION ALL set operator.

Recursive WITH clause complies with the American National Standards Institute (ANSI) standard.

Recursive WITH can be used to query hierarchical data such as organization charts.

Recursive `WITH` Clause: Example

FLIGHTS Table

	SOURCE	DESTIN	FLIGHT_TIME
1	San Jose	Los Angeles	1.3
2	New York	Boston	1.1
3	Los Angeles	New York	5.8

1

```
WITH Reachable_From (Source, Destin, TotalFlightTime) AS
(
    SELECT Source, Destin, Flight_time
    FROM Flights
  UNION ALL
    SELECT incoming.Source, outgoing.Destin,
           incoming.TotalFlightTime+outgoing.Flight_time
    FROM Reachable_From incoming, Flights outgoing
    WHERE incoming.Destin = outgoing.Source
)
SELECT Source, Destin, TotalFlightTime
FROM Reachable_From;
```

2

	SOURCE	DESTIN	TOTALFLIGHTTIME
1	San Jose	Los Angeles	1.3
2	New York	Boston	1.1
3	Los Angeles	New York	5.8
4	Los Angeles	Boston	6.9
5	San Jose	New York	7.1
6	San Jose	Boston	8.2

3

ORACLE

Example 1 in the slide displays records from a `FLIGHTS` table describing flights between two cities.

Using the query in example 2, you query the `FLIGHTS` table to display the total flight time between any source and destination. The `WITH` clause in the query, which is named `Reachable From`, has a `UNION ALL` query with two branches. The first branch is the *anchor* branch, which selects all the rows from the `Flights` table. The second branch is the recursive branch. It joins the contents of `Reachable From` to the `Flights` table to find other cities that can be reached, and adds these to the content of `Reachable From`. The operation will finish when no more rows are found by the recursive branch.

Example 3 displays the result of the query that selects everything from the `WITH` clause element `Reachable From`.

For details, see:

- *Oracle Database SQL Language Reference 12c Release 1.0*
- *Oracle Database Data Warehousing Guide 12c Release 1.0*

Quiz

With a correlated subquery, the inner `SELECT` statement drives the outer `SELECT` statement.

a. True

b. False

Answer: b

Summary

In this lesson, you should have learned how to:

- Write a multiple-column subquery
- Use scalar subqueries in SQL
- Solve problems with correlated subqueries
- Use the `EXISTS` and `NOT EXISTS` operators
- Use the `WITH` clause

You can use multiple-column subqueries to combine multiple `WHERE` conditions in a single `WHERE` clause. Column comparisons in a multiple-column subquery can be pairwise comparisons or nonpairwise comparisons.

You can use a subquery to define a table to be operated on by a containing query.

Scalar subqueries can be used in:

- The condition and expression part of `DECODE` and `CASE`
- All clauses of `SELECT` except `GROUP BY`
- A `SET` clause and `WHERE` clause of the `UPDATE` statement

The Oracle Server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a `SELECT` statement. Using the `WITH` clause, you can reuse the same query when it is costly to re-evaluate the query block and it occurs more than once within a complex query.

Practice 6: Overview

This practice covers the following topics:

- Creating multiple-column subqueries
- Writing correlated subqueries
- Using the `EXISTS` operator
- Using scalar subqueries
- Using the `WITH` clause

In this practice, you write multiple-column subqueries, and correlated and scalar subqueries. You also solve problems by writing the `WITH` clause.

Manipulating Data by Using Subqueries

Objectives

After completing this lesson, you should be able to:

- Use subqueries to manipulate data
- Insert values by using a subquery as a target
- Use the `WITH CHECK OPTION` keyword on DML statements
- Use correlated subqueries to update and delete rows

In this lesson, you learn how to manipulate data in the Oracle database by using subqueries. You also learn how to solve problems by using correlated subqueries.

Lesson Agenda

- Using subqueries to manipulate data
- Inserting values by using a subquery as a target
- Using the `WITH CHECK OPTION` keyword on DML statements
- Using correlated subqueries to update and delete rows

Using Subqueries to Manipulate Data

You can use subqueries in data manipulation language (DML) statements to:

- Retrieve data by using an inline view
- Copy data from one table to another
- Update data in one table based on the values of another table
- Delete rows from one table based on rows in another table

Subqueries can be used to retrieve data from a table that you can use as input to an `INSERT` into a different table. In this way, you can easily copy large volumes of data from one table to another with one single `SELECT` statement. Similarly, you can use subqueries to do mass updates and deletes by using them in the `WHERE` clause of the `UPDATE` and `DELETE` statements. You can also use subqueries in the `FROM` clause of a `SELECT` statement. This is called an inline view.

Note: You learned how to update and delete rows based on another table in the course titled *Oracle Database: SQL Workshop I*.

Lesson Agenda

- Using subqueries to manipulate data
- Inserting values by using a subquery as a target
- Using the `WITH CHECK OPTION` keyword on DML statements
- Using correlated subqueries to update and delete rows

Inserting by Using a Subquery as a Target

```
INSERT INTO (SELECT l.location_id, l.city, l.country_id
             FROM   loc l
             JOIN   countries c
             ON(l.country_id = c.country_id)
             JOIN regions USING(region_id)
             WHERE region_name = 'Europe')
VALUES (3300, 'Cardiff', 'UK');
```

```
1 rows inserted.
```


You can use a subquery in place of the table name in the `INTO` clause of the `INSERT` statement. The `SELECT` list of this subquery must have the same number of columns as the column list of the `VALUES` clause. Any rules on the columns of the base table must be followed in order for the `INSERT` statement to work successfully. For example, you cannot put in a duplicate location `ID` or leave out a value for a mandatory `NOT NULL` column.

This use of subqueries helps you avoid having to create a view just for performing an `INSERT`.

The example in the slide uses a subquery in the place of `LOC` to create a record for a new European city.

Note: You can also perform the `INSERT` operation on the `EUROPEAN_CITIES` view by using the following code:

```
INSERT INTO european_cities
VALUES (3300,'Cardiff','UK');
```

For the example in the slide, the loc table is created by running the following statement:

```
CREATE TABLE loc AS SELECT * FROM locations;
```

Inserting by Using a Subquery as a Target

Verify the results.

```
SELECT location_id, city, country_id
FROM   loc;
```

20	2900	Geneva	CH
21	3000	Bern	CH
22	3100	Utrecht	NL
23	3200	Mexico City	MX
24	3300	Cardiff	UK

The example in the slide shows that the insert via the inline view created a new record in the base table `LOC`.

The following example shows the results of the subquery that was used to identify the table for the `INSERT` statement.

```
SELECT l.location_id, l.city, l.country_id
FROM   loc l
JOIN   countries c
ON(l.country_id = c.country_id)
JOIN regions USING(region_id)
WHERE region_name = 'Europe';
```

Lesson Agenda

- Using subqueries to manipulate data
- Inserting values by using a subquery as a target
- Using the `WITH CHECK OPTION` keyword on DML statements
- Using correlated subqueries to update and delete rows

Using the WITH CHECK OPTION Keyword on DML Statements

The `WITH CHECK OPTION` keyword prohibits you from changing rows that are not in the subquery.

```
INSERT INTO ( SELECT location_id, city, country_id
              FROM   loc
              WHERE  country_id IN
              (SELECT country_id
               FROM countries
               NATURAL JOIN regions
               WHERE region_name = 'Europe')
               WITH CHECK OPTION )
VALUES (3600, 'Washington', 'US');
```

Error report:
SQL Error: ORA-01402: view WITH CHECK OPTION where-clause violation
01402. 00000 - "view WITH CHECK OPTION where-clause violation"
***Cause:**
***Action:**

Specify the `WITH CHECK OPTION` keyword to indicate that if the subquery is used in place of a table in an `INSERT`, `UPDATE`, or `DELETE` statement, no changes that will produce rows that are not included in the subquery are permitted to that table.

The example in the slide shows how to use an inline view with `WITH CHECK OPTION`. The `INSERT` statement prevents the creation of records in the `LOC` table for a city that is not in Europe.

The following example executes successfully because of the changes in the `VALUES` list.

```
INSERT INTO (SELECT location_id, city, country_id
             FROM   loc
             WHERE  country_id IN
             (SELECT country_id
              FROM countries
              NATURAL JOIN regions
              WHERE region_name = 'Europe')
              WITH CHECK OPTION)
VALUES (3500, 'Berlin', 'DE');
```

The use of an inline view with the `WITH CHECK OPTION` provides an easy method to prevent changes to the table.

To prevent the creation of a non-European city, you can also use a database view by performing the following steps:

1. Create a database view:

```
CREATE OR REPLACE VIEW european_cities
AS
SELECT location_id, city, country_id
FROM   locations
WHERE  country_id in
  (SELECT country_id
   FROM countries
   NATURAL JOIN regions
   WHERE region_name = 'Europe')
WITH CHECK OPTION;
```

2. Verify the results by inserting data:

```
INSERT INTO european_cities
VALUES (3400,'New York','US');
```

The second step produces the same error as shown in the slide.

Lesson Agenda

- Using subqueries to manipulate data
- Inserting values by using a subquery as a target
- Using the `WITH CHECK OPTION` keyword on DML statements
- Using correlated subqueries to update and delete rows

Correlated UPDATE

Use a correlated subquery to update rows in one table based on rows from another table.

```
UPDATE table1 alias1
SET    column = (SELECT expression
                 FROM   table2 alias2
                 WHERE  alias1.column =
                        alias2.column);
```


In the case of the `UPDATE` statement, you can use a correlated subquery to update rows in one table based on rows from another table.

Using Correlated UPDATE

- Denormalize the EMPL6 table by adding a column to store the department name.
- Populate the table by using a correlated update.

```
ALTER TABLE empl6
ADD(department_name VARCHAR2(25));
```

```
UPDATE empl6 e
SET    department_name =
              (SELECT department_name
               FROM   departments d
               WHERE  e.department_id = d.department_id);
```


The example in the slide denormalizes the EMPL6 table by adding a column to store the department name and then populates the table by using a correlated update.

Following is another example for a correlated update.

Problem Statement

The REWARDS table has a list of employees who have exceeded expectations in their performance. Use a correlated subquery to update rows in the EMPL6 table based on rows from the REWARDS table:

```
UPDATE empl6
SET    salary = (SELECT empl6.salary + rewards.pay_raise
                 FROM   rewards
                 WHERE  employee_id  =
                        empl6.employee_id
                 AND  payraise_date =
                      (SELECT MAX(payraise_date)
                      FROM   rewards
                      WHERE  employee_id = empl6.employee_id))
WHERE  empl6.employee_id
IN     (SELECT employee_id FROM rewards);
```

This example uses the REWARDS table. The REWARDS table has the following columns: EMPLOYEE_ID, PAY_RAISE, and PAYRAISE_DATE. Every time an employee gets a pay raise, a record with details such as the employee ID, the amount of the pay raise, and the date of receipt of the pay raise is inserted into the REWARDS table. The REWARDS table can contain more than one record for an employee. The PAYRAISE _DATE column is used to identify the most recent pay raise received by an employee.

In the example, the SALARY column in the EMPL6 table is updated to reflect the latest pay raise received by the employee. This is done by adding the current salary of the employee with the corresponding pay raise from the REWARDS table.

Correlated `DELETE`

Use a correlated subquery to delete rows in one table based on rows from another table.

```
DELETE FROM table1 alias1
WHERE  column operator
               (SELECT expression
                FROM   table2 alias2
                WHERE  alias1.column = alias2.column);
```


In the case of a `DELETE` statement, you can use a correlated subquery to delete only those rows that also exist in another table. If you decide that you will maintain only the last four job history records in the `JOB_HISTORY` table, when an employee transfers to a fifth job, you delete the oldest `JOB_HISTORY` row by looking up the `JOB_HISTORY` table for the `MIN(START_DATE)` for the employee. The following code illustrates how the preceding operation can be performed using a correlated `DELETE`:

```
DELETE FROM job_history JH
WHERE employee_id =
      (SELECT employee_id
       FROM employees E
       WHERE JH.employee_id = E.employee_id
       AND START_DATE =
             (SELECT MIN(start_date)
              FROM job_history JH
              WHERE JH.employee_id = E.employee_id)
              AND 5 >  (SELECT COUNT(*)
                        FROM job_history JH
                        WHERE JH.employee_id = E.employee_id
                        GROUP BY EMPLOYEE_ID
                        HAVING COUNT(*) >= 4));
```

Using Correlated DELETE

Use a correlated subquery to delete only those rows from the EMPL6 table that also exist in the EMP_HISTORY table.

```
DELETE FROM empl6 E
WHERE employee_id =
           (SELECT employee_id
            FROM   emp_history
            WHERE  employee_id = E.employee_id);
```


Example

Two tables are used in this example. They are:

- The EMPL6 table, which provides details of all the current employees
- The EMP_HISTORY table, which provides details of previous employees

EMP_HISTORY contains data regarding previous employees, so it would be erroneous if the same employee's record existed in both the EMPL6 and EMP_HISTORY tables. You can delete such erroneous records by using the correlated subquery shown in the slide.

Summary

In this lesson, you should have learned how to:

- Manipulate data by using subqueries
- Insert values by using a subquery as a target
- Use the `WITH CHECK OPTION` keyword on DML statements
- Use correlated subqueries with `UPDATE` and `DELETE` statements

In this lesson, you should have learned how to manipulate data in the Oracle database by using subqueries. You learn how to use the `WITH CHECK OPTION` keyword on DML statements and use correlated subqueries with `UPDATE` and `DELETE` statements.

Practice 7: Overview

This practice covers the following topics:

- Using subqueries to manipulate data
- Inserting values by using a subquery as a target
- Using the `WITH CHECK OPTION` keyword on DML statements
- Using correlated subqueries to update and delete rows

In this practice, you learn the concepts of manipulating data by using subqueries, `WITH CHECK OPTION`, and correlated subqueries to `UPDATE` and `DELETE` rows.

8

Controlling User Access

Objectives

After completing this lesson, you should be able to do the following:

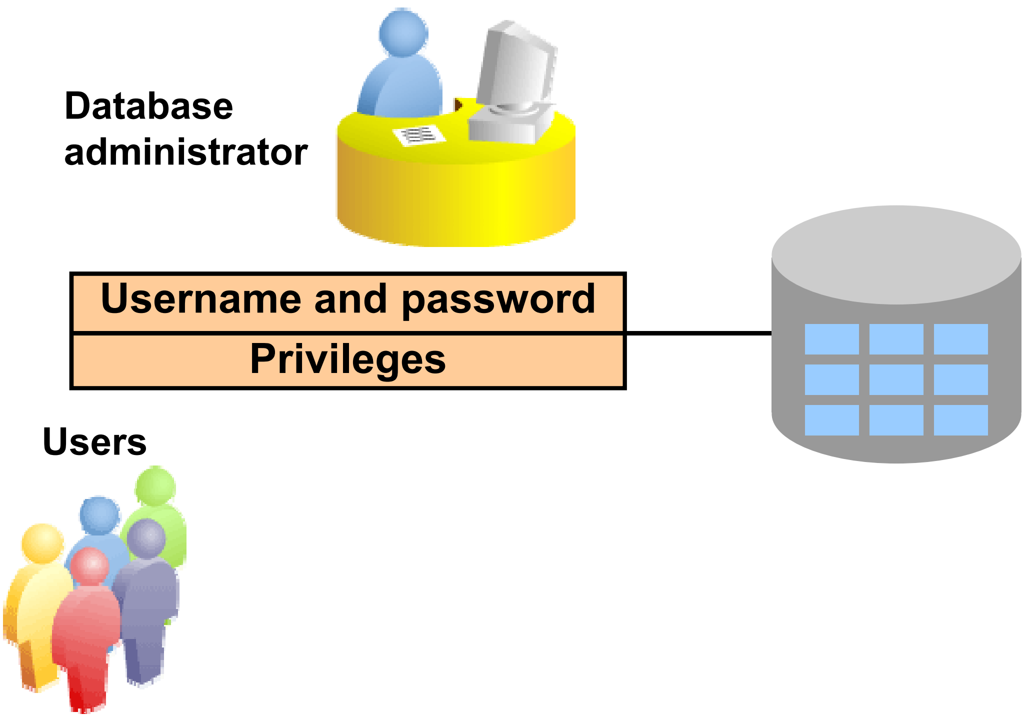
- Differentiate system privileges from object privileges
- Grant privileges on tables
- Grant roles
- Distinguish between privileges and roles

In this lesson, you learn how to control database access to specific objects and add new users with different levels of access privileges.

Lesson Agenda

- System privileges
- Creating a role
- Object privileges
- Revoking object privileges

Controlling User Access



In a multiple-user environment, you want to maintain security of database access and use. With Oracle Server database security, you can do the following:

- Control database access.
- Give access to specific objects in the database.
- Confirm given and received privileges with the Oracle data dictionary.

Database security can be classified into two categories: system security and data security. System security covers access and use of the database at the system level, such as the username and password, the disk space allocated to users, and the system operations that users can perform. Database security covers access and use of the database objects and the actions that those users can perform on the objects.

Privileges

- Database security:
  - System security
  - Data security
- System privileges: Performing a particular action within the database
- Object privileges: Manipulating the content of the database objects
- Schemas: Collection of objects such as tables, views, and sequences

A privilege is the right to execute particular SQL statements. The database administrator (DBA) is a high-level user with the ability to create users and grant users access to the database and its objects. Users require *system privileges* to gain access to the database and *object privileges* to manipulate the content of the objects in the database. Users can also be given the privilege to grant additional privileges to other users or to *roles*, which are named groups of related privileges.

Schemas

A *schema* is a collection of objects such as tables, views, and sequences. The schema is owned by a database user and has the same name as that user.

A system privilege is the right to perform a particular action, or to perform an action on any schema objects of a particular type. An object privilege provides the user the ability to perform a particular action on a specific schema object.

For more information, see the *Oracle Database 2 Day DBA* reference manual for Oracle Database12*c*.

System Privileges

- More than 200 privileges are available.
- The database administrator has high-level system privileges for tasks such as:
  - Creating new users
  - Removing users
  - Removing tables
  - Backing up tables

ORACLE

More than 200 distinct system privileges are available for users and roles. Typically, system privileges are provided by the database administrator (DBA).

The table `SYSTEM_PRIVILEGE_MAP` contains all the system privileges available, based on the version release. This table is also used to map privilege type numbers to type names.

Typical DBA Privileges

System Privilege	**Operations Authorized**
`CREATE USER`	Grantee can create other Oracle users.
`DROP USER`	Grantee can drop another user.
`DROP ANY TABLE`	Grantee can drop a table in any schema.
`BACKUP ANY TABLE`	Grantee can back up any table in any schema with the export utility.
`SELECT ANY TABLE`	Grantee can query tables, views, or materialized views in any schema.
`CREATE ANY TABLE`	Grantee can create tables in any schema.

Creating Users

The DBA creates users with the `CREATE USER` statement.

```
CREATE USER user
IDENTIFIED BY   password;
```

```
CREATE USER  demo
IDENTIFIED BY demo;
```


The DBA creates the user by executing the `CREATE USER` statement. The user does not have any privileges at this point. The DBA can then grant privileges to that user. These privileges determine what the user can do at the database level.

The slide gives the abridged syntax for creating a user.

In the syntax:

user	Is the name of the user to be created
Password	Specifies that the user must log in with this password

For more information, see the *Oracle Database SQL Language Reference* for Oracle Database12*c*.

Note: Starting with Oracle Database 11*g*, passwords are case-sensitive.

User System Privileges

- After a user is created, the DBA can grant specific system privileges to that user.

```
GRANT privilege [, privilege...]
TO user [, user| role, PUBLIC...];
```

- An application developer, for example, may have the following system privileges:
  - `CREATE SESSION`
  - `CREATE TABLE`
  - `CREATE SEQUENCE`
  - `CREATE VIEW`
  - `CREATE PROCEDURE`

Typical User Privileges

After the DBA creates a user, the DBA can assign privileges to that user.

System Privilege	Operations Authorized
`CREATE SESSION`	Connect to the database.
`CREATE TABLE`	Create tables in the user's schema.
`CREATE SEQUENCE`	Create a sequence in the user's schema.
`CREATE VIEW`	Create a view in the user's schema.
`CREATE PROCEDURE`	Create a stored procedure, function, or package in the user's schema.

In the syntax:

privilege	Is the system privilege to be granted
user |role|PUBLIC	Is the name of the user, the name of the role, or PUBLIC (which designates that every user is granted the privilege)

Note: Current system privileges can be found in the SESSION_PRIVS dictionary view. Data dictionary is a collection of tables and views created and maintained by the Oracle Server. They contain information about the database.

Granting System Privileges

The DBA can grant specific system privileges to a user.

```
GRANT  create session, create table,
       create sequence, create view
TO     demo;
```

```
GRANT succeeded.
```

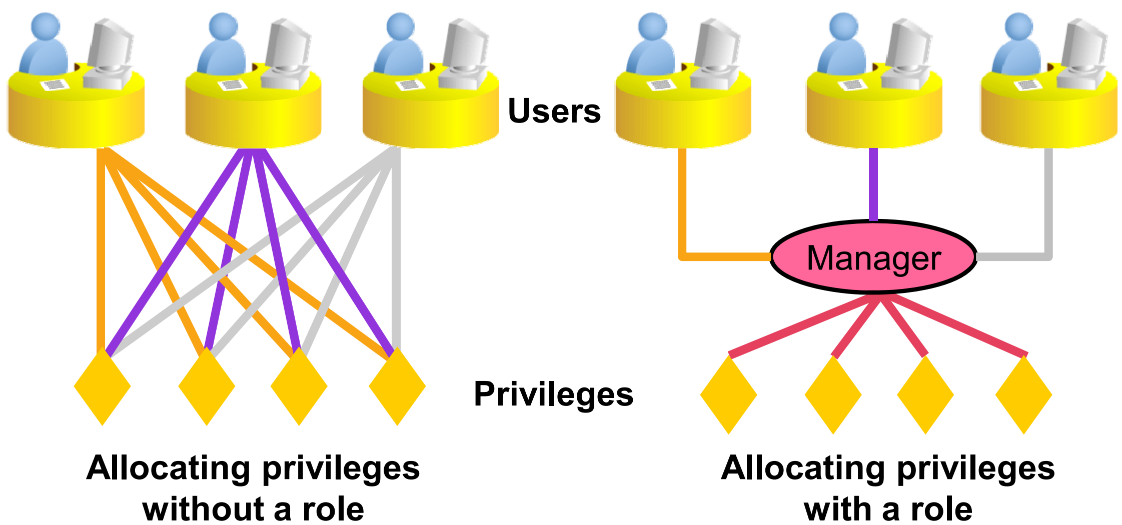

The DBA uses the `GRANT` statement to allocate system privileges to the user. After the user has been granted the privileges, the user can immediately use those privileges.

In the example in the slide, the `demo` user has been assigned the privileges to create sessions, tables, sequences, and views.

Lesson Agenda

- System privileges
- Creating a role
- Object privileges
- Revoking object privileges

What Is a Role?



A role is a named group of related privileges that can be granted to the user. This method makes it easier to revoke and maintain privileges.

A user can have access to several roles, and several users can be assigned the same role. Roles are typically created for a database application.

Creating and Assigning a Role

First, the DBA must create the role. Then the DBA can assign privileges to the role and assign the role to users.

Syntax

```
CREATE   ROLE role;
```

In the syntax:

role Is the name of the role to be created

After the role is created, the DBA can use the `GRANT` statement to assign the role to users as well as assign privileges to the role. A role is not a schema object; therefore, any user can add privileges to a role.

Creating and Granting Privileges to a Role

- Create a role:

```
CREATE ROLE manager;
```

- Grant privileges to a role:

```
GRANT create table, create view
TO manager;
```

- Grant a role to users:

```
GRANT manager TO alice;
```


Creating a Role

The example in the slide creates a `manager` role and then enables the manager to create tables and views. It then grants user `alice` the role of a manager. Now `alice` can create tables and views.

If users have multiple roles granted to them, they receive all the privileges associated with all the roles.

Changing Your Password

- The DBA creates your user account and initializes your password.
- You can change your password by using the `ALTER USER` statement.

```
ALTER USER demo
IDENTIFIED BY employ;
```


The DBA creates an account and initializes a password for every user. You can change your password by using the `ALTER USER` statement.

The slide example shows that the `demo` user changes the password by using the `ALTER USER` statement.

Syntax

```
ALTER USER user IDENTIFIED BY password;
```

In the syntax:

user	Is the name of the user
password	Specifies the new password

Although this statement can be used to change your password, there are many other options. You must have the `ALTER USER` privilege to change any other option.

For more information, see the *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

Note: SQL*Plus has a `PASSWORD` command (`PASSW`) that can be used to change the password of a user when the user is logged in. This command is not available in SQL Developer.

Lesson Agenda

- System privileges
- Creating a role
- Object privileges
- Revoking object privileges

Object Privileges

Object privilege	Table	View	Sequence
ALTER	✓		✓
DELETE	✓	✓	
INDEX	✓		
INSERT	✓	✓	
REFERENCES	✓		
SELECT	✓	✓	✓
UPDATE	✓	✓	

An *object privilege* is a privilege or right to perform a particular action on a specific table, view, sequence, or procedure. Each object has a particular set of grantable privileges. The table in the slide lists the privileges for various objects. Note that the only privileges that apply to a sequence are `SELECT` and `ALTER`. `UPDATE`, `REFERENCES`, and `INSERT` can be restricted by specifying a subset of updatable columns.

A `SELECT` privilege can be restricted by creating a view with a subset of columns and granting the `SELECT` privilege only on the view. A privilege granted on a synonym is converted to a privilege on the base table referenced by the synonym.

Note: With the `REFERENCES` privilege, you can ensure that other users can create `FOREIGN KEY` constraints that reference your table.

Object Privileges

- Object privileges vary from object to object.
- An owner has all the privileges on the object.
- An owner can give specific privileges on that owner's object.

```
GRANT      object_priv [(columns)]
ON         object
TO         {user|role|PUBLIC}
[WITH GRANT OPTION];
```


Granting Object Privileges

Different object privileges are available for different types of schema objects. A user automatically has all object privileges for schema objects contained in the user's schema. A user can grant any object privilege on any schema object that the user owns to any other user or role. If the grant includes `WITH GRANT OPTION`, the grantee can further grant the object privilege to other users; otherwise, the grantee can use the privilege but cannot grant it to other users.

In the syntax:

object_priv	Is an object privilege to be granted
`ALL`	Specifies all object privileges
columns	Specifies the column from a table or view on which privileges are granted
`ON` *object*	Is the object on which the privileges are granted
`TO`	Identifies to whom the privilege is granted
`PUBLIC`	Grants object privileges to all users
`WITH GRANT OPTION`	Enables the grantee to grant the object privileges to other users and roles

Note: In the syntax, *schema* is the same as the owner's name.

Granting Object Privileges

- Grant query privileges on the `EMPLOYEES` table:

```
GRANT  select
ON     employees
TO     demo;
```

- Grant privileges to update specific columns to users and roles:

```
GRANT  update (department_name, location_id)
ON     departments
TO     demo, manager;
```


Guidelines

- To grant privileges on an object, the object must be in your own schema, or you must have been granted the object privileges `WITH GRANT OPTION`.
- An object owner can grant any object privilege on the object to any other user or role of the database.
- The owner of an object automatically acquires all object privileges on that object.

The first example in the slide grants the `demo` user the privilege to query your `EMPLOYEES` table. The second example grants `UPDATE` privileges on specific columns in the `DEPARTMENTS` table to `demo` and to the `manager` role.

For example, if your schema is `oraxx`, and the `demo` user now wants to use a `SELECT` statement to obtain data from your `EMPLOYEES` table, the syntax he or she must use is:

```
SELECT  * FROM oraxx.employees;
```

Alternatively, the `demo` user can create a synonym for the table and issue a `SELECT` statement from the synonym:

```
CREATE SYNONYM emp FOR oraxx.employees;
SELECT * FROM emp;
```

Note: DBAs generally allocate system privileges; any user who owns an object can grant object privileges.

Passing On Your Privileges

- Give a user authority to pass along privileges:

```
GRANT   select, insert
ON      departments
TO      demo
WITH    GRANT OPTION;
```

- Allow all users on the system to query data from `DEPARTMENTS` table:

```
GRANT   select
ON      departments
TO      PUBLIC;
```

ORACLE

`WITH GRANT OPTION` Keyword

A privilege that is granted with the `WITH GRANT OPTION` clause can be passed on to other users and roles by the grantee. Object privileges granted with the `WITH GRANT OPTION` clause are revoked when the grantor's privilege is revoked. You can specify `WITH GRANT OPTION` only when granting to a user or to `PUBLIC`, not when granting to a role.

The grantor must meet one or more of the below criteria. The grantor:

- Must be the object owner; otherwise, the grantor must have object access with `GRANT OPTION` from the user
- Must have the `GRANT ANY OBJECT PRIVILEGE` system privilege and an object privilege on the object

The example in the slide gives the `demo` user access to your `DEPARTMENTS` table with the privileges to query the table and add rows to the table. The example also shows that `demo` can give others these privileges.

`PUBLIC` Keyword

An owner of a table can grant access to all users by using the `PUBLIC` keyword. The second example allows all users on the system to query data from the `DEPARTMENTS` table.

Confirming Granted Privileges

Data Dictionary View	Description
`ROLE_SYS_PRIVS`	System privileges granted to roles
`ROLE_TAB_PRIVS`	Table privileges granted to roles
`USER_ROLE_PRIVS`	Roles accessible by the user
`USER_SYS_PRIVS`	System privileges granted to the user
`USER_TAB_PRIVS_MADE`	Object privileges granted on the user's objects
`USER_TAB_PRIVS_RECD`	Object privileges granted to the user
`USER_COL_PRIVS_MADE`	Object privileges granted on the columns of the user's objects
`USER_COL_PRIVS_RECD`	Object privileges granted to the user on specific columns

ORACLE

If you attempt to perform an unauthorized operation, such as deleting a row from a table for which you do not have the `DELETE` privilege, the Oracle server does not permit the operation to take place.

If you receive the Oracle server error message "Table or view does not exist," you have done either of the following:

- Named a table or view that does not exist
- Attempted to perform an operation on a table or view for which you do not have the appropriate privilege

The data dictionary is organized in tables and views and contains information about the database. You can access the data dictionary to view the privileges that you have. The table in the slide describes various data dictionary views.

You learn about data dictionary views in the lesson titled "Introduction to Data Dictionary Views."

Note: The `ALL_TAB_PRIVS_MADE` dictionary view describes all the object grants made by the user or made on the objects owned by the user.

Lesson Agenda

- System privileges
- Creating a role
- Object privileges
- Revoking object privileges

Revoking Object Privileges

- You use the `REVOKE` statement to revoke privileges granted to other users.
- Privileges granted to others through the `WITH GRANT OPTION` clause are also revoked.

```
REVOKE {privilege [, privilege...]|ALL}
ON     object
FROM   {user[, user...]|role|PUBLIC}
[CASCADE CONSTRAINTS];
```


You can remove privileges granted to other users by using the `REVOKE` statement. When you use the `REVOKE` statement, the privileges that you specify are revoked from the users you name and from any other users to whom those privileges were granted by the revoked user.

In the syntax:

`CASCADE`	Is required to remove any referential integrity constraints made to the `CONSTRAINTS` object by means of the `REFERENCES` privilege

For more information, see the *Oracle Database SQL Language Reference* for Oracle Database12*c*.

Note: If a user leaves the company and you revoke his or her privileges, you must regrant any privileges that this user granted to other users. If you drop the user account without revoking privileges from it, the system privileges granted by this user to other users are not affected by this action.

Revoking Object Privileges

Revoke the `SELECT` and `INSERT` privileges given to the `demo` user on the `DEPARTMENTS` table.

```
REVOKE  select, insert
ON      departments
FROM    demo;
```

```
REVOKE succeeded.
```


The example in the slide revokes `SELECT` and `INSERT` privileges given to the `demo` user on the `DEPARTMENTS` table.

Note: If a user is granted a privilege with the `WITH GRANT OPTION` clause, that user can also grant the privilege with the `WITH GRANT OPTION` clause, so that a long chain of grantees is possible, but no circular grants (granting to a grant ancestor) are permitted. If the owner revokes a privilege from a user who granted the privilege to other users, the revoking cascades to all the privileges granted.

For example, if user `A` grants a `SELECT` privilege on a table to user `B` including the `WITH GRANT OPTION` clause, user `B` can grant to user `C` the `SELECT` privilege with the `WITH GRANT OPTION` clause as well, and user `C` can then grant to user `D` the `SELECT` privilege. If user `A` revokes privileges from user `B`, the privileges granted to users `C` and `D` are also revoked.

Quiz

Which of the following statements are true?

a. After a user creates an object, the user can pass along any of the available object privileges to other users by using the `GRANT` statement.
b. A user can create roles by using the `CREATE ROLE` statement to pass along a collection of system or object privileges to other users.
c. Users can change their own passwords.
d. Users can view the privileges granted to them and those that are granted on their objects.

Answer: a, c, d

Summary

In this lesson, you should have learned how to:

- Differentiate system privileges from object privileges
- Grant privileges on tables
- Grant roles
- Distinguish between privileges and roles

DBAs establish initial database security for users by assigning privileges to the users.

- The DBA creates users who must have a password. The DBA is also responsible for establishing the initial system privileges for a user.
- After the user has created an object, the user can pass along any of the available object privileges to other users or to all users by using the `GRANT` statement.
- A DBA can create roles by using the `CREATE ROLE` statement to pass along a collection of system or object privileges to multiple users. Roles make granting and revoking privileges easier to maintain.
- Users can change their passwords by using the `ALTER USER` statement.
- You can remove privileges from users by using the `REVOKE` statement.
- With data dictionary views, users can view the privileges granted to them and those that are granted on their objects.

Practice 8: Overview

This practice covers the following topics:

- Granting other users privileges to your table
- Modifying another user's table through the privileges granted to you

ORACLE

In this practice, you learn how to grant other users privileges to your table and modifying another user's table through the privileges granted to you.

Manipulating Data

Objectives

After completing this lesson, you should be able to:

- Specify explicit default values in the `INSERT` and `UPDATE` statements
- Describe the features of multitable `INSERT`s
- Use the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merge rows in a table
- Perform flashback operations
- Track the changes made to data over a period of time

In this lesson, you learn how to use the `DEFAULT` keyword in `INSERT` and `UPDATE` statements to identify a default column value. You also learn about multitable `INSERT` statements, the `MERGE` statement, performing flashback operations, and tracking changes in the database.

Lesson Agenda

- Specifying explicit default values in the `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Performing flashback operations
- Tracking the changes to data over a period of time

Explicit Default Feature: Overview

- Use the `DEFAULT` keyword as a column value where the default column value is desired.
- This allows the user to control where and when the default value should be applied to data.
- Explicit defaults can be used in `INSERT` and `UPDATE` statements.

The `DEFAULT` keyword can be used in `INSERT` and `UPDATE` statements to identify a default column value. If no default value exists, a null value is used.

The `DEFAULT` option saves you from having to hard code the default value in your programs or query the dictionary to find it, as was done before this feature was introduced. Hard-coding the default is a problem if the default changes, because the code consequently needs changing. Accessing the dictionary is not usually done in an application; therefore, this is a very important feature.

Using Explicit Default Values

- `DEFAULT` with `INSERT`:

```
INSERT INTO deptm3
  (department_id, department_name, manager_id)
VALUES (300, 'Engineering', DEFAULT);
```

- `DEFAULT` with `UPDATE`:

```
UPDATE deptm3
SET manager_id = DEFAULT
WHERE department_id = 10;
```

ORACLE

Specify `DEFAULT` to set the column to the value previously specified as the default value for the column. If no default value for the corresponding column has been specified, the Oracle server sets the column to null.

In the first example in the slide, the `INSERT` statement uses a default value for the `MANAGER_ID` column. If there is no default value defined for the column, a null value is inserted instead.

The second example uses the `UPDATE` statement to set the `MANAGER_ID` column to a default value for department 10. If no default value is defined for the column, it changes the value to null.

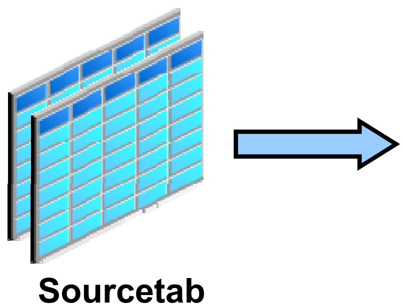
Note: When creating a table, you can specify a default value for a column. This is discussed in *SQL Workshop I*.

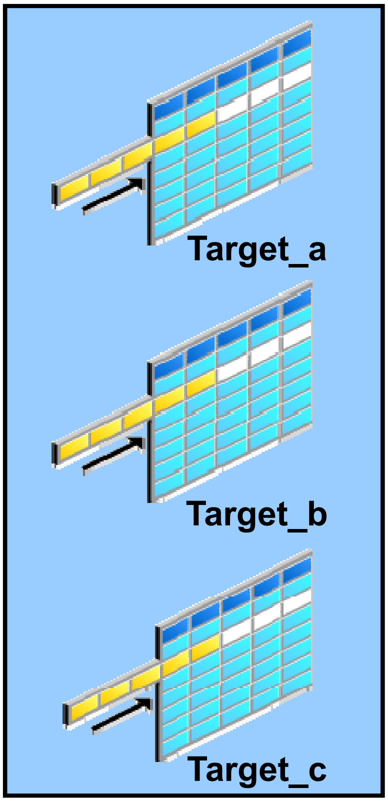
Lesson Agenda

- Specifying explicit default values in the `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Performing flashback operations
- Tracking the changes to data over a period of time

ORACLE

Multitable `INSERT` Statements: Overview






```
INSERT  ALL
   INTO target_a VALUES(…,…,…)
   INTO target_b VALUES(…,…,…)
   INTO target_c VALUES(…,…,…)
   SELECT …
   FROM  sourcetab
   WHERE …;
```

Subquery

In a multitable `INSERT` statement, you insert computed rows derived from the rows returned from the evaluation of a subquery into one or more tables.

Multitable `INSERT` statements are useful in a data warehouse scenario. You need to load your data warehouse regularly so that it can serve its purpose of facilitating business analysis. To do this, data from one or more operational systems must be extracted and copied into the warehouse. The process of extracting data from the source system and bringing it into the data warehouse is commonly called ETL, which stands for extraction, transformation, and loading.

During extraction, the desired data must be identified and extracted from many different sources, such as database systems and applications. After extraction, the data must be physically transported to the target system or an intermediate system for further processing. Depending on the chosen means of transportation, some transformations can be done during this process. For example, a SQL statement that directly accesses a remote target through a gateway can concatenate two columns as part of the `SELECT` statement.

After data is loaded into the Oracle database, data transformations can be executed using SQL operations. A multitable `INSERT` statement is one of the techniques for implementing SQL data transformations.

Multitable `INSERT` Statements: Overview

- Use the `INSERT…SELECT` statement to insert rows into multiple tables as part of a single DML statement.
- Multitable `INSERT` statements are used in data warehousing systems to transfer data from one or more operational sources to a set of target tables.
- They provide significant performance improvement over:
  - Single DML versus multiple `INSERT…SELECT` statements
  - Single DML versus a procedure to perform multiple inserts by using the `IF...THEN` syntax

Multitable `INSERT` statements offer the benefits of the `INSERT ... SELECT` statement when multiple tables are involved as targets. Without multitable `INSERT`, you had to deal with *n* independent `INSERT ... SELECT` statements, thus processing the same source data *n* times and increasing the transformation workload *n* times.

As with the existing `INSERT ... SELECT` statement, the new statement can be parallelized and used with the direct-load mechanism for faster performance.

Each record from any input stream, such as a nonrelational database table, can now be converted into multiple records for a more relational database table environment. To alternatively implement this functionality, you were required to write multiple `INSERT` statements.

Types of Multitable `INSERT` Statements

The different types of multitable `INSERT` statements are:

- Unconditional `INSERT`
- Conditional `INSERT ALL`
- Conditional `INSERT FIRST`
- Pivoting `INSERT`

You use different clauses to indicate the type of `INSERT` to be executed. The types of multitable `INSERT` statements are:

- **Unconditional `INSERT`:** For each row returned by the subquery, a row is inserted into each of the target tables.
- **Conditional `INSERT ALL`:** For each row returned by the subquery, a row is inserted into each target table if the specified condition is met.
- **Conditional `INSERT FIRST`:** For each row returned by the subquery, a row is inserted into the very first target table in which the condition is met.
- **Pivoting `INSERT`:** This is a special case of the unconditional `INSERT ALL`.

Multitable INSERT Statements

- Syntax for multitable INSERT:

```
{ ALL
{ insert_into_clause [ values_clause ]}...
| conditional_insert_clause
 } subquery
```

- conditional_insert_clause:

```
[ ALL | FIRST ]
 WHEN condition THEN insert_into_clause
                               [ values_clause ]
                               [ insert_into_clause [ values_clause ]]…
  [ ELSE insert_into_clause
                               [ values_clause ]
                               [ insert_into_clause [ values_clause ]]…
 ]
```

ORACLE

The slide displays the generic format for multitable INSERT statements.

Unconditional INSERT: ALL into_clause

Specify ALL followed by multiple insert_into_clauses to perform an unconditional multitable INSERT. The Oracle Server executes each insert_into_clause once for each row returned by the subquery.

Conditional INSERT: conditional_insert_clause

Specify the conditional_insert_clause to perform a conditional multitable INSERT. The Oracle Server filters each insert_into_clause through the corresponding WHEN condition, which determines whether that insert_into_clause is executed. A single multitable INSERT statement can contain up to 127 WHEN clauses.

Conditional INSERT: ALL

If you specify ALL, the Oracle Server evaluates each WHEN clause regardless of the results of the evaluation of any other WHEN clause. For each WHEN clause whose condition evaluates to true, the Oracle Server executes the corresponding INTO clause list.

Conditional `INSERT: FIRST`

If you specify `FIRST`, the Oracle Server evaluates each `WHEN` clause in the order in which it appears in the statement. If the first `WHEN` clause evaluates to true, the Oracle Server executes the corresponding `INTO` clause and skips subsequent `WHEN` clauses for the given row.

Conditional `INSERT: ELSE` Clause

For a given row, if no `WHEN` clause evaluates to true:

- If you have specified an `ELSE` clause, the Oracle Server executes the `INTO` clause list associated with the `ELSE` clause
- If you did not specify an `ELSE` clause, the Oracle Server takes no action for that row

Restrictions on Multitable `INSERT` Statements

- You can perform multitable `INSERT` statements only on tables, and not on views or materialized views.
- You cannot perform a multitable `INSERT` on a remote table.
- You cannot specify a table collection expression when performing a multitable `INSERT`.
- In a multitable `INSERT`, all `insert_into_clauses` cannot combine to specify more than 999 target columns.

Unconditional INSERT ALL

- Select the EMPLOYEE_ID, HIRE_DATE, SALARY, and MANAGER_ID values from the EMPLOYEES table for those employees whose EMPLOYEE_ID is greater than 200.
- Insert these values into the SAL_HISTORY and MGR_HISTORY tables by using a multitable INSERT.

```
INSERT  ALL
   INTO sal_history VALUES(EMPID,HIREDATE,SAL)
   INTO mgr_history VALUES(EMPID,MGR,SAL)
   SELECT employee_id EMPID, hire_date HIREDATE,
          salary SAL, manager_id MGR
   FROM  employees
   WHERE employee_id > 200;
```

```
12 rows inserted
```


The example in the slide inserts rows into both the SAL_HISTORY and the MGR_HISTORY tables.

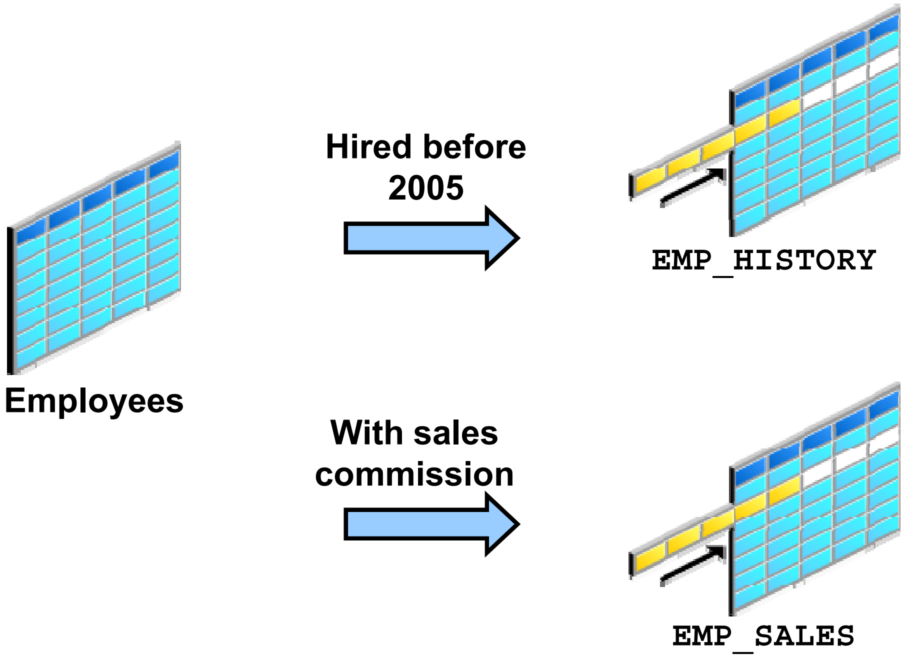
The SELECT statement retrieves the details of employee ID, hire date, salary, and manager ID of those employees whose employee ID is greater than 200 from the EMPLOYEES table. The details of the employee ID, hire date, and salary are inserted into the SAL_HISTORY table. The details of employee ID, manager ID, and salary are inserted into the MGR_HISTORY table.

This INSERT statement is referred to as an unconditional INSERT because no further restriction is applied to the rows that are retrieved by the SELECT statement. All the rows retrieved by the SELECT statement are inserted into the two tables: SAL_HISTORY and MGR_HISTORY. The VALUES clause in the INSERT statements specifies the columns from the SELECT statement that must be inserted into each of the tables. Each row returned by the SELECT statement results in two insertions: one for the SAL_HISTORY table and one for the MGR_HISTORY table.

A total of 12 rows were inserted:

```
SELECT COUNT(*) total_in_sal FROM sal_history;
SELECT COUNT(*) total_in_mgr FROM mgr_history;
```

Conditional INSERT ALL: Example



For all employees in the employees tables, if the employee was hired before 2005, insert that employee record into the employee history. If the employee earns a sales commission, insert the record information into the `EMP_SALES` table. The SQL statement is shown on the next page.

Conditional INSERT ALL

```
INSERT  ALL
 WHEN HIREDATE < '01-JAN-05' THEN
   INTO emp_history VALUES(EMPID,HIREDATE,SAL)
 WHEN COMM IS NOT NULL THEN
   INTO emp_sales VALUES(EMPID,COMM,SAL)
   SELECT employee_id EMPID, hire_date HIREDATE,
          salary SAL, commission_pct COMM
   FROM  employees;
```

```
59 rows inserted.
```


The example in the slide is similar to the example in the previous slide because it inserts rows into both the EMP_HISTORY and the EMP_SALES tables. The SELECT statement retrieves details such as employee ID, hire date, salary, and commission percentage for all employees from the EMPLOYEES table. Details such as employee ID, hire date, and salary are inserted into the EMP_HISTORY table. Details such as employee ID, commission percentage, and salary are inserted into the EMP_SALES table.

This INSERT statement is referred to as a conditional INSERT ALL because a further restriction is applied to the rows that are retrieved by the SELECT statement. From the rows that are retrieved by the SELECT statement, only those rows in which the hire date was prior to 2005 are inserted in the EMP_HISTORY table. Similarly, only those rows where the value of commission percentage is not null are inserted in the EMP_SALES table.

```
SELECT count(*) FROM emp_history;
```

Result: 24 rows fetched.

```
SELECT count(*) FROM emp_sales;
```

Result: 35 rows fetched.

You can also optionally use the ELSE clause with the INSERT ALL statement.

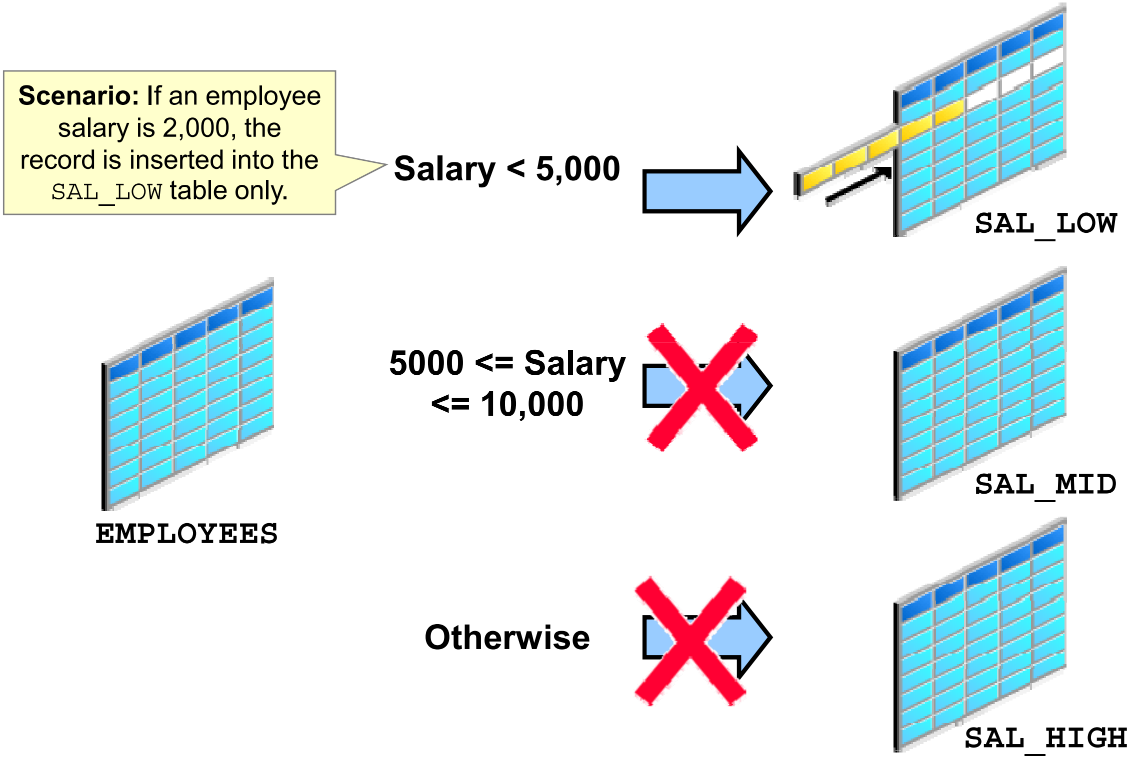
Example:

```
INSERT ALL
WHEN job_id IN
(select job_id FROM jobs WHERE job_title LIKE '%Manager%') THEN
INTO managers2(last_name,job_id,SALARY)
VALUES (last_name,job_id,SALARY)
WHEN SALARY>10000 THEN
INTO richpeople(last_name,job_id,SALARY)
VALUES (last_name,job_id,SALARY)
ELSE
INTO poorpeople VALUES (last_name,job_id,SALARY)
SELECT * FROM employees;
```

Result:

```
116 rows inserted
```

Conditional INSERT FIRST: Example



For all employees in the EMPLOYEES table, insert the employee information into the first target table that meets the condition. In the example, if an employee has a salary of 2,000, the record is inserted into the SAL_LOW table only. The SQL statement is shown on the next page.

Conditional INSERT FIRST

```
INSERT FIRST
WHEN salary < 5000 THEN
  INTO sal_low VALUES (employee_id, last_name, salary)
WHEN salary between 5000 and 10000 THEN
  INTO sal_mid VALUES (employee_id, last_name, salary)
ELSE
  INTO sal_high VALUES (employee_id, last_name, salary)
SELECT employee_id, last_name, salary
FROM employees;
```

```
107 rows inserted
```

ORACLE

The `SELECT` statement retrieves details such as employee `ID`, last name, and salary for every employee in the `EMPLOYEES` table. For each employee record, it is inserted into the very first target table that meets the condition.

This `INSERT` statement is referred to as a conditional `INSERT FIRST`. The `WHEN salary < 5000` condition is evaluated first. If this first `WHEN` clause evaluates to true, the Oracle Server executes the corresponding `INTO` clause and inserts the record into the `SAL_LOW` table. It skips subsequent `WHEN` clauses for this row.

If the row does not satisfy the first `WHEN` condition (`WHEN salary < 5000`), the next condition (`WHEN salary between 5000 and 10000`) is evaluated. If this condition evaluates to true, the record is inserted into the `SAL_MID` table, and the last condition is skipped.

If neither the first condition (`WHEN salary < 5000`) nor the second condition (`WHEN salary between 5000 and 10000`) is evaluated to true, the Oracle Server executes the corresponding `INTO` clause for the `ELSE` clause.

A total of 107 rows were inserted:

```
SELECT count(*) low FROM sal_low;

49 rows fetched.



SELECT count(*) mid FROM sal_mid;

43 rows fetched.



SELECT count(*) high FROM sal_high;

15 rows fetched.
```

Pivoting INSERT

Convert the set of sales records from the nonrelational database table to relational format.

Emp_ID	Week_ID	MON	TUES	WED	THUR	FRI
176	6	2000	3000	4000	5000	6000

Employee_ID	WEEK	SALES
176	6	2000
176	6	3000
176	6	4000
176	6	5000
176	6	6000

Pivoting is an operation in which you must build a transformation such that each record from any input stream, such as a nonrelational database table, must be converted into multiple records for a more relational database table environment.

Suppose you receive a set of sales records from a nonrelational database table:

`SALES_SOURCE_DATA`, in the following format:

`EMPLOYEE_ID, WEEK_ID, SALES_MON, SALES_TUE, SALES_WED, SALES_THUR, SALES_FRI`

You want to store these records in the `SALES_INFO` table in a more typical relational format:

`EMPLOYEE_ID, WEEK, SALES`

To solve this problem, you must build a transformation such that each record from the original nonrelational database table, `SALES_SOURCE_DATA`, is converted into five records for the data warehouse's `SALES_INFO` table. This operation is commonly referred to as *pivoting*.

The solution to this problem is shown on the next page.

Pivoting INSERT

```
INSERT ALL
  INTO sales_info VALUES (employee_id,week_id,sales_MON)
  INTO sales_info VALUES (employee_id,week_id,sales_TUE)
  INTO sales_info VALUES (employee_id,week_id,sales_WED)
  INTO sales_info VALUES (employee_id,week_id,sales_THUR)
  INTO sales_info VALUES (employee_id,week_id, sales_FRI)
  SELECT EMPLOYEE_ID, week_id, sales_MON, sales_TUE,
         sales_WED, sales_THUR,sales_FRI
  FROM sales_source_data;
```

```
5 rows inserted
```


In the example in the slide, the sales data is received from the nonrelational database table `SALES_SOURCE_DATA`, which is the details of the sales performed by a sales representative on each day of a week, for a week with a particular week `ID`.

`DESC SALES_SOURCE_DATA`

```
DESC SALES_SOURCE_DATA
Name        Null Type
----------- ---- -----------
EMPLOYEE_ID      NUMBER(6)
WEEK_ID          NUMBER(2)
SALES_MON        NUMBER(8,2)
SALES_TUE        NUMBER(8,2)
SALES_WED        NUMBER(8,2)
SALES_THUR       NUMBER(8,2)
SALES_FRI        NUMBER(8,2)
```

```
SELECT * FROM SALES_SOURCE_DATA;
```

	EMPLOYEE_ID	WEEK_ID	SALES_MON	SALES_TUE	SALES_WED	SALES_THUR	SALES_FRI
1	178	6	1750	2200	1500	1500	3000

```
DESC SALES_INFO
```

```
desc sales_info
Name        Null Type
----------- ---- -----------
EMPLOYEE_ID      NUMBER(6)
WEEK             NUMBER(2)
SALES            NUMBER(8,2)
```

```
SELECT * FROM sales_info;
```

	EMPLOYEE_ID	WEEK	SALES
1	178	6	1750
2	178	6	2200
3	178	6	1500
4	178	6	1500
5	178	6	3000

Observe in the preceding example that by using a pivoting `INSERT`, one row from the `SALES_SOURCE_DATA` table is converted into five records for the relational table, `SALES_INFO`.

Lesson Agenda

- Specifying explicit default values in the `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERTs`:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Performing flashback operations
- Tracking the changes to data over a period of time

ORACLE

`MERGE` Statement

- Provides the ability to conditionally update, insert, or delete data into a database table
- Performs an `UPDATE` if the row exists, and an `INSERT` if it is a new row:
  - Avoids separate updates
  - Increases performance and ease of use
  - Is useful in data warehousing applications

The Oracle Server supports the `MERGE` statement for `INSERT`, `UPDATE`, and `DELETE` operations. Using this statement, you can update, insert, or delete a row conditionally into a table, thus avoiding multiple DML statements. The decision whether to update, insert, or delete into the target table is based on a condition in the `ON` clause.

You must have the `INSERT` and `UPDATE` object privileges on the target table and the `SELECT` object privilege on the source table. To specify the `DELETE` clause of `merge_update_clause`, you must also have the `DELETE` object privilege on the target table.

The `MERGE` statement is deterministic. You cannot update the same row of the target table multiple times in the same `MERGE` statement.

An alternative approach is to use PL/SQL loops and multiple DML statements. The `MERGE` statement, however, is easy to use and more simply expressed as a single SQL statement.

The `MERGE` statement is suitable in a number of data warehousing applications. For example, in a data warehousing application, you may need to work with data coming from multiple sources, some of which may be duplicates. With the `MERGE` statement, you can conditionally add or modify rows.

MERGE Statement Syntax

You can conditionally insert, update, or delete rows in a table by using the MERGE statement.

```
MERGE INTO table_name table_alias
  USING (table|view|sub_query) alias
  ON (join condition)
  WHEN MATCHED THEN
    UPDATE SET
    col1 = col1_val,
    col2 = col2_val
 WHEN NOT MATCHED THEN
    INSERT (column_list)
    VALUES (column_values);
```

ORACLE

Merging Rows

You can update existing rows, and insert new rows conditionally by using the MERGE statement. Using the MERGE statement, you can delete obsolete rows at the same time as you update rows in a table. To do this, you include a DELETE clause with its own WHERE clause in the syntax of the MERGE statement.

In the syntax:

INTO clause	Specifies the target table you are updating or inserting into
USING clause	Identifies the source of the data to be updated or inserted; can be a table, view, or subquery
ON clause	The condition on which the MERGE operation either updates or inserts
WHEN MATCHED | WHEN NOT MATCHED	Instructs the server how to respond to the results of the join condition

Note: For more information, see *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

Merging Rows: Example

Insert or update rows in the `COPY_EMP3` table to match the `EMPLOYEES` table.

```
MERGE INTO copy_emp3 c
USING (SELECT * FROM EMPLOYEES ) e
ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
UPDATE SET
c.first_name = e.first_name,
c.last_name = e.last_name,
...

DELETE WHERE (E.COMMISSION_PCT IS NOT NULL)
WHEN NOT MATCHED THEN
INSERT VALUES(e.employee_id, e.first_name, e.last_name,
e.email, e.phone_number, e.hire_date, e.job_id,
e.salary, e.commission_pct, e.manager_id,
e.department_id);
```

```
107 rows merged.
```



```
MERGE INTO copy_emp3 c
USING (SELECT * FROM EMPLOYEES ) e
ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
UPDATE SET
c.first_name = e.first_name,
c.last_name = e.last_name,
c.email = e.email,
c.phone_number = e.phone_number,
c.hire_date = e.hire_date,
c.job_id = e.job_id,
c.salary = e.salary*2,
c.commission_pct = e.commission_pct,
c.manager_id = e.manager_id,
c.department_id = e.department_id
DELETE WHERE (E.COMMISSION_PCT IS NOT NULL)
WHEN NOT MATCHED THEN
```

```
INSERT VALUES(e.employee_id, e.first_name, e.last_name,
e.email, e.phone_number, e.hire_date, e.job_id,
e.salary, e.commission_pct, e.manager_id,
e.department_id);
```

The `COPY_EMP3` table is created by using the following code:

```
CREATE TABLE COPY_EMP3 AS SELECT * FROM EMPLOYEES
WHERE SALARY<10000;
```

Then query the `COPY_EMP3` table.

```
SELECT employee_id, salary, commission_pct FROM COPY_EMP3;
```

Observe that in the output, there are some employees with `SALARY < 10000` and there are two employees with `COMMISSION_PCT`.

The example in the slide matches the `EMPLOYEE_ID` in the `COPY_EMP3` table to the `EMPLOYEE_ID` in the `EMPLOYEES` table. If a match is found, the row in the `COPY_EMP3` table is updated to match the row in the `EMPLOYEES` table and the salary of the employee is doubled. The records of the two employees with values in the `COMMISSION_PCT` column are deleted. If the match is not found, rows are inserted into the `COPY_EMP3` table.

Merging Rows: Example

```
TRUNCATE TABLE copy_emp3;
SELECT * FROM copy_emp3;
no rows selected
```

```
MERGE INTO copy_emp3 c
USING (SELECT * FROM EMPLOYEES ) e
ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
UPDATE SET
c.first_name = e.first_name,
c.last_name = e.last_name,
...
DELETE WHERE (E.COMMISSION_PCT IS NOT NULL)
WHEN NOT MATCHED THEN
INSERT VALUES(e.employee_id, e.first_name, ...
```

```
SELECT * FROM copy_emp3;
107 rows selected.
```


The examples in the slide show that the `COPY_EMP3` table is empty. The `c.employee_id = e.employee_id` condition is evaluated. The condition returns false—there are no matches. The logic falls into the `WHEN NOT MATCHED` clause, and the `MERGE` command inserts the rows of the `EMPLOYEES` table into the `COPY_EMP3` table. This means that the `COPY_EMP3` table now has exactly the same data as in the `EMPLOYEES` table.

```
SELECT employee_id, salary, commission_pct from copy_emp3;
```

Lesson Agenda

- Specifying explicit default values in the `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERTs`:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Performing flashback operations
- Tracking the changes to data over a period of time

`FLASHBACK TABLE` Statement

- Enables you to recover tables to a specified point in time with a single statement
- Restores table data along with associated indexes and constraints
- Enables you to revert the table and its contents to a certain point in time or system change number (SCN)

Oracle Flashback Table enables you to recover tables to a specified point in time with a single statement. You can restore table data along with associated indexes and constraints while the database is online, undoing changes to only the specified tables.

The Flashback Table feature is similar to a self-service repair tool. For example, if a user accidentally deletes important rows from a table and then wants to recover the deleted rows, you can use the `FLASHBACK TABLE` statement to restore the table to the time before the deletion and see the missing rows in the table.

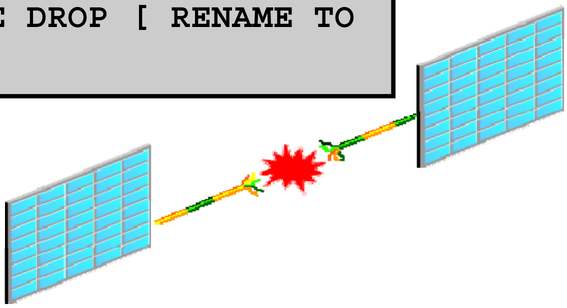
When using the `FLASHBACK TABLE` statement, you can revert the table and its contents to a certain time or to an SCN.

Note: The SCN is an integer value associated with each change to the database. It is a unique incremental number in the database. Every time you commit a transaction, a new SCN is recorded.

FLASHBACK TABLE Statement

- Repair tool for accidental table modifications
  - Restores a table to an earlier point in time
  - Offers ease of use, availability, and fast execution
  - Is performed in place
- Syntax:

```
FLASHBACK TABLE [ schema. ] table [, [ schema.
] table ]... TO { { { SCN | TIMESTAMP } expr |
RESTORE POINT restore_point } [ { ENABLE |
DISABLE } TRIGGERS ] | BEFORE DROP [ RENAME TO
table ] } ;
```



Self-Service Repair Facility

Oracle Database provides a SQL data definition language (DDL) command, `FLASHBACK TABLE`, to restore the state of a table to an earlier point in time in case it is inadvertently deleted or modified. The `FLASHBACK TABLE` command is a self-service repair tool to restore data in a table along with associated attributes such as indexes or views. This is done, while the database is online, by rolling back only the subsequent changes to the given table. Compared to traditional recovery mechanisms, this feature offers significant benefits such as ease of use, availability, and faster restoration. It also takes the burden off the DBA to find and restore application-specific properties. The flashback table feature does not address physical corruption caused because of a bad disk.

Syntax

You can invoke a `FLASHBACK TABLE` operation on one or more tables, even on tables in different schemas. You specify the point in time to which you want to revert by providing a valid time stamp. By default, database triggers are disabled during the flashback operation for all tables involved. You can override this default behavior by specifying the `ENABLE TRIGGERS` clause.

Note: For more information about recycle bin and flashback semantics, refer to *Oracle Database Administrator's Guide* for Oracle Database 12*c*.

Using the FLASHBACK TABLE Statement

```
DROP TABLE emp3;
```

```
table EMP3 dropped.
```

```
SELECT original_name, operation, droptime FROM
recyclebin;
```

	ORIGINAL_NAME	OPERATION	DROPTIME
1	EMP3	DROP	2012-10-16:05:59:34

…

```
FLASHBACK TABLE emp3 TO BEFORE DROP;
```

```
table EMP3 succeeded.
```


Syntax and Examples

The example restores the `EMP3` table to a state before a `DROP` statement.

The recycle bin is actually a data dictionary table containing information about dropped objects. Dropped tables and any associated objects—such as, indexes, constraints, nested tables, and so on—are not removed and still occupy space. They continue to count against user space quotas until specifically purged from the recycle bin, or until they must be purged by the database because of tablespace space constraints.

Each user can be thought of as an owner of a recycle bin because, unless a user has the `SYSDBA` privilege, the only objects that the user has access to in the recycle bin are those that the user owns. A user can view his or her objects in the recycle bin by using the following statement:

```
SELECT * FROM RECYCLEBIN;
```

When you drop a user, any objects belonging to that user are not placed in the recycle bin and any objects in the recycle bin are purged.

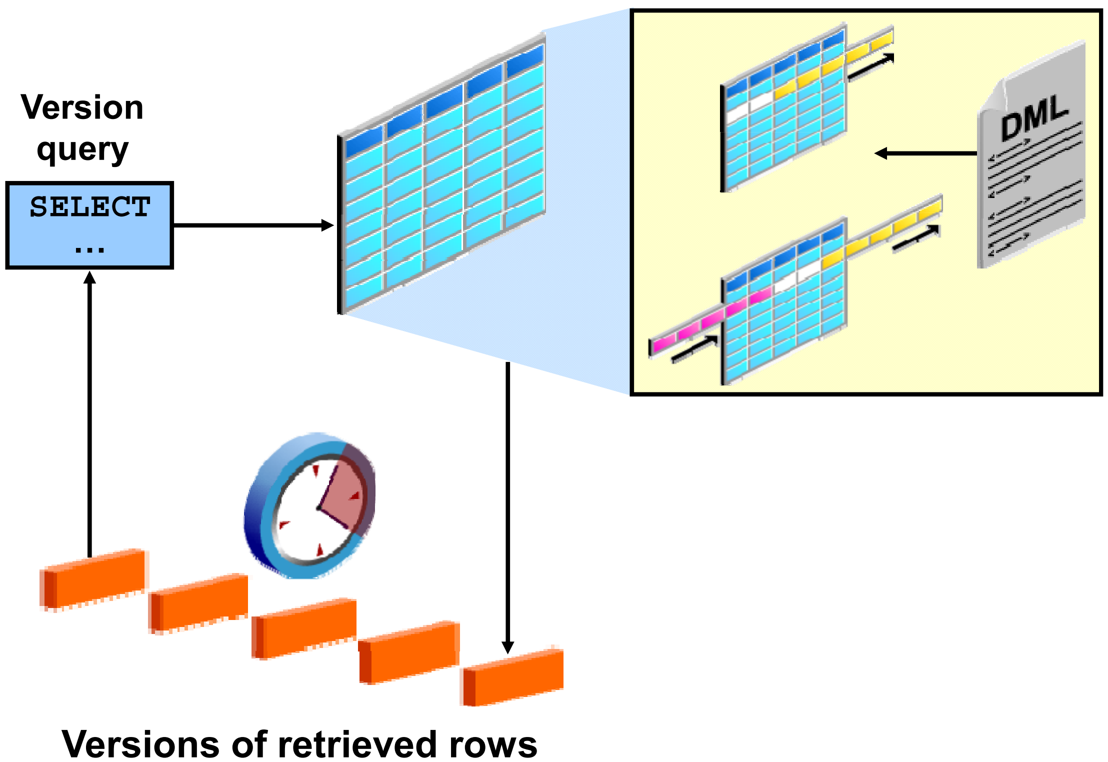
You can purge the recycle bin with the following statement:

```
PURGE RECYCLEBIN;
```

Lesson Agenda

- Specifying explicit default values in the `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERTs`:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Performing flashback operations
- Tracking the changes to data over a period of time

Tracking Changes in Data



You may discover that, somehow, data in a table has been inappropriately changed. To research this, you can use multiple flashback queries to view row data at specific points in time. You can use Oracle Flashback Query to retrieves data as it existed at an earlier time. More efficiently, you can use the Flashback Version Query feature to view all changes to a row over a period of time. This feature enables you to append a `VERSIONS` clause to a `SELECT` statement that specifies a system change number (SCN) or the time stamp range within which you want to view changes to row values. The query also can return associated metadata, such as the transaction responsible for the change.

Further, after you identify an erroneous transaction, you can use the Flashback Transaction Query feature to identify other changes that were done by the transaction. You then have the option of using the Flashback Table feature to restore the table to a state before the changes were made.

You can use a query on a table with a `VERSIONS` clause to produce all the versions of all the rows that exist, or ever existed, between the time the query was issued and the `undo_retention` seconds before the current time. `undo_retention` is an initialization parameter, which is an autotuned parameter. A query that includes a `VERSIONS` clause is referred to as a version query. The results of a version query behaves as though the `WHERE` clause were applied to the versions of the rows. The version query returns versions of the rows only across transactions.

System change number (SCN): The Oracle server assigns an SCN to identify the redo records for each committed transaction.

Flashback Query: Example

```
SELECT salary FROM employees3
WHERE last_name = 'Chung';
```

```
UPDATE employees3 SET salary = 4000
WHERE last_name = 'Chung';

SELECT salary FROM employees3
WHERE last_name = 'Chung';
```

```
SELECT salary FROM employees3
AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' MINUTE)
WHERE last_name = 'Chung';
```

1

	SALARY
1	3800

2

	SALARY
1	4000

3

	SALARY
1	3800

To use Oracle Flashback Query, use a `SELECT` statement with an `AS OF` clause. Oracle Flashback Query retrieves data as it existed at some time in the past. The query explicitly references a past time through a timestamp or System Change Number (SCN). It returns committed data that was current at that point in time.

In the example in the slide, the salary for employee Chung is retrieved (1). The salary for employee Chung is increased to 4000 (2). To learn what the value was before the update, you can use the Flashback Query(3).

Oracle Flashback Query can be used in the following scenarios:

- Recovering lost data or undoing incorrect, committed changes. For example, if you mistakenly delete or update rows, and then commit them, you can immediately undo the mistake.
- Comparing current data with the corresponding data at some time in the past. For example, you can run a daily report that shows the change in data from yesterday. You can compare individual rows of table data or find intersections or unions of sets of rows.
- Checking the state of transactional data at a particular time

Flashback Version Query: Example

```
SELECT salary FROM employees3
WHERE  employee_id = 107;
```

1

```
UPDATE employees3 SET salary = salary * 1.30
WHERE  employee_id = 107;

COMMIT;
```

2

```
SELECT salary FROM employees3
  VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
WHERE  employee_id = 107;
```

3

1

	SALARY
1	4200

3

	SALARY
1	5460
2	4200

In the example in the slide, the salary for employee 107 is retrieved (1). The salary for employee 107 is increased by 30 percent and this change is committed (2). The different versions of salary are displayed (3).

The `VERSIONS` clause does not change the plan of the query. For example, if you run a query on a table that uses the index access method, the same query on the same table with a `VERSIONS` clause continues to use the index access method. The versions of the rows returned by the version query are versions of the rows across transactions. The `VERSIONS` clause has no effect on the transactional behavior of a query. This means that a query on a table with a `VERSIONS` clause still inherits the query environment of the ongoing transaction.

The default `VERSIONS` clause can be specified as `VERSIONS BETWEEN {SCN|TIMESTAMP} MINVALUE AND MAXVALUE`. The `VERSIONS` clause is a SQL extension only for queries. You can have DML and DDL operations that use a `VERSIONS` clause within subqueries. The row version query retrieves all the committed versions of the selected rows. Changes made by the current active transaction are not returned. The version query retrieves all incarnations of the rows. This essentially means that versions returned include deleted and subsequent reinserted versions of the rows. The row access for a version query can be defined in one of the following two categories:

- **ROWID-based row access:** In case of `ROWID`-based access, all versions of the specified `ROWID` are returned irrespective of the row content. This essentially means that all versions of the slot in the block indicated by the `ROWID` are returned.
- **All other row access:** For all other row access, all versions of the rows are returned.

VERSIONS BETWEEN Clause

```
SELECT versions_starttime "START_DATE",
       versions_endtime   "END_DATE",
       salary
FROM   employees
    VERSIONS BETWEEN SCN MINVALUE
    AND MAXVALUE
WHERE  last_name = 'Lorentz';
```

	START_DATE	END_DATE	SALARY
1	11-SEP-12 03.38.54.000000000 AM	(null)	5460
2	(null)	11-SEP-12 03.38.54.000000000 AM	4200

```
SELECT salary FROM employees3
  VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
WHERE  employee_id = 107;
```


You can use the VERSIONS BETWEEN clause to retrieve all the versions of the rows that exist or have ever existed between the time the query was issued and a point back in time.

If the undo retention time is less than the lower bound time or the SCN of the BETWEEN clause, the query retrieves versions up to the undo retention time only. The time interval of the BETWEEN clause can be specified as an SCN interval or a wall-clock interval. This time interval is closed at both the lower and the upper bounds.

In the example, Lorentz's salary changes are retrieved. The NULL value for END_DATE for the first version indicates that this was the existing version at the time of the query. The NULL value for START_DATE for the last version indicates that this version was created at a time before the undo retention time.

Quiz

When you use the `INSERT` or `UPDATE` command, the `DEFAULT` keyword saves you from hard-coding the default value in your programs or querying the dictionary to find it.

a. True

b. False

Answer: a

Quiz

In all the cases, when you execute a `DROP TABLE` command, the database renames the table and places it in a recycle bin, from where it can later be recovered by using the `FLASHBACK TABLE` statement.

a. True

b. False

Answer: b

Summary

In this lesson, you should have learned how to:

- Specify explicit default values in the `INSERT` and `UPDATE` statements
- Describe the features of multitable `INSERT`s
- Use the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merge rows in a table
- Perform flashback operations
- Track the changes to data over a period of time

In this lesson, you also should have learned about multitable `INSERT` statements, the `MERGE` statement, and tracking changes in the database.

Practice 9: Overview

This practice covers the following topics:

- Performing multitable `INSERT`s
- Performing `MERGE` operations
- Performing flashback operations
- Tracking row versions

In this practice, you learn how to perform multitable `INSERTS`, `MERGE` operations, flashback operations, and tracking row versions.

10

Managing Data in Different Time Zones

ORACLE

Objectives

After completing this lesson, you should be able to:

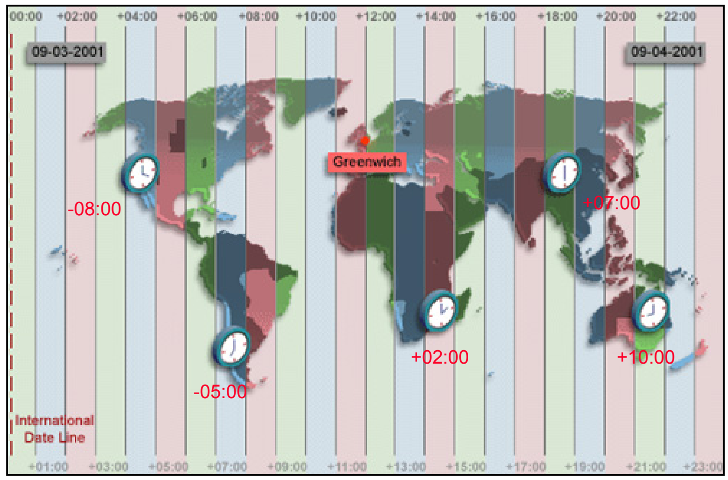
- Use data types similar to `DATE` that store fractional seconds and track time zones
- Use data types that store the difference between two datetime values
- Use the following datetime functions:
  - `CURRENT_DATE`
  - `CURRENT_TIMESTAMP`
  - `LOCALTIMESTAMP`
  - `DBTIMEZONE`
  - `SESSIONTIMEZONE`
  - `EXTRACT`
  - `TZ_OFFSET`
  - `FROM_TZ`
  - `TO_TIMESTAMP`
  - `TO_YMINTERVAL`
  - `TO_DSINTERVAL`

In this lesson, you learn how to use data types similar to `DATE` that store fractional seconds and track time zones. This lesson addresses some of the datetime functions available in the Oracle database.

Lesson Agenda

- CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP
- INTERVAL data types
- Using the following functions:
  - EXTRACT
  - TZ_OFFSET
  - FROM_TZ
  - TO_TIMESTAMP
  - TO_YMINTERVAL
  - TO_DSINTERVAL

Time Zones



The image represents the time for each time zone when Greenwich time is 12:00.

The hours of the day are measured by the turning of the earth. The time of day at any particular moment depends on where you are. When it is noon in Greenwich, England, it is midnight along the International Date Line. The earth is divided into 24 time zones, one for each hour of the day. The time along the prime meridian in Greenwich, England, is known as Greenwich Mean Time (GMT). GMT is now known as Coordinated Universal Time (UTC). UTC is the time standard against which all other time zones in the world are referenced. It is the same all year round and is not affected by summer time or daylight saving time. The meridian line is an imaginary line that runs from the North Pole to the South Pole. It is known as zero longitude and it is the line from which all other lines of longitude are measured. All time is measured relative to UTC and all places have a latitude (their distance north or south of the equator) and a longitude (their distance east or west of the Greenwich meridian).

`TIME_ZONE` Session Parameter

`TIME_ZONE` may be set to:

- An absolute offset
- Database time zone
- OS local time zone
- A named region

```
ALTER SESSION SET TIME_ZONE = '-05:00';
ALTER SESSION SET TIME_ZONE = dbtimezone;
ALTER SESSION SET TIME_ZONE = local;
ALTER SESSION SET TIME_ZONE = 'America/New_York';
```


The Oracle database supports storing the time zone in your date and time data, as well as fractional seconds. The `ALTER SESSION` command can be used to change time zone values in a user’s session. The time zone values can be set to an absolute offset, a named time zone, a database time zone, or the local time zone.

CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP

- CURRENT_DATE:
  – Returns the current date from the user session
  – Has a data type of DATE
- CURRENT_TIMESTAMP:
  – Returns the current date and time from the user session
  – Has a data type of TIMESTAMP WITH TIME ZONE
- LOCALTIMESTAMP:
  – Returns the current date and time from the user session
  – Has a data type of TIMESTAMP

The CURRENT_DATE and CURRENT_TIMESTAMP functions return the current date and current time stamp, respectively. The data type of CURRENT_DATE is DATE. The data type of CURRENT_TIMESTAMP is TIMESTAMP WITH TIME ZONE. The values returned display the time zone displacement of the SQL session executing the functions. The time zone displacement is the difference (in hours and minutes) between local time and UTC. The TIMESTAMP WITH TIME ZONE data type has the format:

```
TIMESTAMP [ (fractional_seconds_precision) ] WITH TIME ZONE
```

where fractional_seconds_precision optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 through 9. The default is 6.

The LOCALTIMESTAMP function returns the current date and time in the session time zone. The difference between LOCALTIMESTAMP and CURRENT_TIMESTAMP is that LOCALTIMESTAMP returns a TIMESTAMP value, whereas CURRENT_TIMESTAMP returns a TIMESTAMP WITH TIME ZONE value.

These functions are national language support (NLS)–sensitive—that is, the results will be in the current NLS calendar and datetime formats.

Note: The SYSDATE function returns the current date and time as a DATE data type. You learned how to use the SYSDATE function in the course titled *Oracle Database: SQL Workshop I*.

Comparing Date and Time in a Session's Time Zone

The TIME_ZONE parameter is set to –5:00 and then SELECT statements for each date and time are executed to compare differences.

```
ALTER SESSION
SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
ALTER SESSION SET TIME_ZONE = '-5:00';

SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;          1


SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP FROM DUAL;     2


SELECT SESSIONTIMEZONE, LOCALTIMESTAMP FROM DUAL;        3
```

ORACLE

The ALTER SESSION command sets the date format of the session to 'DD-MON-YYYY HH24:MI:SS'–that is, day of month (1–31)-abbreviated name of month-4-digit year hour of day (0–23):minute (0–59):second (0–59).

The example in the slide illustrates that the session is altered to set the TIME_ZONE parameter to –5:00. Then the SELECT statement for CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP is executed to observe the differences in format.

Note: The TIME_ZONE parameter specifies the default local time zone displacement for the current SQL session. TIME_ZONE is a session parameter only, not an initialization parameter. The TIME_ZONE parameter is set as follows:

```
TIME_ZONE = '[+ | -] hh:mm'
```

The format mask ([+ | -] hh:mm) indicates the hours and minutes before or after UTC.

Comparing Date and Time in a Session's Time Zone

Results of queries:

```
session SET altered.
```

1

	SESSIONTIMEZONE	CURRENT_DATE
1	-05:00	10-SEP-2012 23:08:16

2

	SESSIONTIMEZONE	CURRENT_TIMESTAMP
1	-05:00	10-SEP-12 11.10.21.183775000 PM -05:00

3

	SESSIONTIMEZONE	LOCALTIMESTAMP
1	-05:00	10-SEP-12 11.10.43.871626000 PM

In this case, the `CURRENT_DATE` function returns the current date in the session's time zone, the `CURRENT_TIMESTAMP` function returns the current date and time in the session's time zone as a value of the data type `TIMESTAMP WITH TIME ZONE`, and the `LOCALTIMESTAMP` function returns the current date and time in the session's time zone.

Note: The code example output may vary depending on when the command is run.

DBTIMEZONE and SESSIONTIMEZONE

- Display the value of the database time zone:

```
SELECT DBTIMEZONE FROM DUAL;
```

	DBTIMEZONE
1	+00:00

- Display the value of the session's time zone:

```
SELECT SESSIONTIMEZONE FROM DUAL;
```

	SESSIONTIMEZONE
1	-05:00

The DBA sets the database's default time zone by specifying the `SET TIME_ZONE` clause of the `CREATE DATABASE` statement. If omitted, the default database time zone is the operating system time zone. The database time zone cannot be changed for a session with an `ALTER SESSION` statement.

The `DBTIMEZONE` function returns the value of the database time zone. The return type is a time zone offset (a character type in the format: `'[+|-]TZH:TZM'`) or a time zone region name, depending on how the user specified the database time zone value in the most recent `CREATE DATABASE` or `ALTER DATABASE` statement. The example in the slide shows that the database time zone is set to "–05:00," as the `TIME_ZONE` parameter is in the format:

```
TIME_ZONE = '[+ | -] hh:mm'
```

The `SESSIONTIMEZONE` function returns the value of the current session's time zone. The return type is a time zone offset (a character type in the format `'[+|-]TZH:TZM'`) or a time zone region name, depending on how the user specified the session time zone value in the most recent `ALTER SESSION` statement. The example in the slide shows that the session time zone is offset to UTC by –5 hours. Observe that the database time zone is different from the current session's time zone.

TIMESTAMP Data Types

Data Type	Fields
TIMESTAMP	Year, Month, Day, Hour, Minute, Second with fractional seconds
TIMESTAMP WITH TIME ZONE	Same as the TIMESTAMP data type; also includes: TIMEZONE_HOUR, and TIMEZONE_MINUTE or TIMEZONE_REGION
TIMESTAMP WITH LOCAL TIME ZONE	Same as the TIMESTAMP data type; also includes a time zone offset in its value

The TIMESTAMP data type is an extension of the DATE data type.

TIMESTAMP (fractional_seconds_ precision)

This data type contains the year, month, and day values of date, as well as hour, minute, and second values of time, where significant fractional seconds precision is the number of digits in the fractional part of the SECOND datetime field. The accepted values of significant fractional_seconds_precision are 0 through 9. The default is 6.

TIMESTAMP (fractional_seconds_precision) WITH TIME ZONE

This data type contains all values of TIMESTAMP as well as time zone displacement value.

TIMESTAMP (fractional_seconds_precision) WITH LOCAL TIME ZONE

This data type contains all values of TIMESTAMP, with the following exceptions:

- Data is normalized to the database time zone when it is stored in the database.
- When the data is retrieved, users see the data in the session time zone.

TIMESTAMP Fields

Datetime Field	Valid Values
YEAR	–4712 to 9999 (excluding year 0)
MONTH	01 to 12
DAY	01 to 31
HOUR	00 to 23
MINUTE	00 to 59
SECOND	00 to 59.9(N) where 9(N) is precision
TIMEZONE_HOUR	–12 to 14
TIMEZONE_MINUTE	00 to 59

Each datetime data type is composed of several of these fields. Datetimes are mutually comparable and assignable only if they have the same datetime fields.

Difference Between DATE and TIMESTAMP

A

```
-- when hire_date is
of type DATE

SELECT hire_date
FROM emp4;
```

	HIRE_DATE
1	17-JUN-03
2	21-SEP-05
3	13-JAN-01
4	03-JAN-06
5	21-MAY-07
6	25-JUN-05
7	05-FEB-06
8	07-FEB-07
9	17-AUG-02
10	16-AUG-02

…

B

```
ALTER TABLE emp4
MODIFY hire_date
TIMESTAMP(7);
SELECT hire_date
FROM emp4;
```

	HIRE_DATE
1	17-JUN-03 12.00.00.000000000 AM
2	21-SEP-05 12.00.00.000000000 AM
3	13-JAN-01 12.00.00.000000000 AM
4	03-JAN-06 12.00.00.000000000 AM
5	21-MAY-07 12.00.00.000000000 AM
6	25-JUN-05 12.00.00.000000000 AM
7	05-FEB-06 12.00.00.000000000 AM
8	07-FEB-07 12.00.00.000000000 AM
9	17-AUG-02 12.00.00.000000000 AM
10	16-AUG-02 12.00.00.000000000 AM

…

ORACLE

TIMESTAMP Data Type: Example

In the slide, example A shows the data from the hire_date column of the EMP4 table when the data type of the column is DATE. In example B, the table is altered and the data type of the hire_date column is made into TIMESTAMP. The output shows the differences in display. You can convert from DATE to TIMESTAMP when the column has data, but you cannot convert from DATE or TIMESTAMP to TIMESTAMP WITH TIME ZONE unless the column is empty.

You can specify the fractional seconds precision for time stamp. If none is specified, as in this example, it defaults to 6.

For example, the following statement sets the fractional seconds precision as 7:

```
ALTER TABLE emp4
MODIFY hire_date TIMESTAMP(7);
```

Note: The Oracle DATE data type by default looks like what is shown in this example. However, the date data type also contains additional information such as hours, minutes, seconds, AM, and PM. To obtain the date in this format, you can apply a format mask or a function to the date value.

Comparing `TIMESTAMP` Data Types

```
CREATE TABLE web_orders
(order_date TIMESTAMP WITH TIME ZONE,
 delivery_time TIMESTAMP WITH LOCAL TIME ZONE);
```

```
INSERT INTO web_orders values
(current_date, current_timestamp + 2);
```

```
SELECT * FROM web_orders;
```

	ORDER_DATE	DELIVERY_TIME
1	10-SEP-12 11.30.20.000000000 PM -05:00	12-SEP-12 11.30.20.000000000 PM

In the example in the slide, a new table `web_orders` is created with a column of data type `TIMESTAMP WITH TIME ZONE` and a column of data type `TIMESTAMP WITH LOCAL TIME ZONE`. This table is populated whenever a `web_order` is placed. The time stamp and time zone for the user placing the order are inserted based on the `CURRENT_DATE` value. The local time stamp and time zone are populated by inserting two days from the `CURRENT_TIMESTAMP` value into it every time an order is placed. When a web-based company guarantees shipping, it can estimate its delivery time based on the time zone of the person placing the order.

Note: The code example output may vary as per the time of run of the command.

Lesson Agenda

- CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP
- INTERVAL data types
- Using the following functions:
  - EXTRACT
  - TZ_OFFSET
  - FROM_TZ
  - TO_TIMESTAMP
  - TO_YMINTERVAL
  - TO_DSINTERVAL

INTERVAL Data Types

- INTERVAL data types are used to store the difference between two datetime values.
- There are two classes of intervals:
  - Year-month
  - Day-time
- The precision of the interval is:
  - The actual subset of fields that constitutes an interval
  - Specified in the interval qualifier

Data Type	Fields
INTERVAL YEAR TO MONTH	Year, Month
INTERVAL DAY TO SECOND	Days, Hour, Minute, Second with fractional seconds

INTERVAL data types are used to store the difference between two datetime values. There are two classes of intervals: year-month intervals and day-time intervals. A year-month interval is made up of a contiguous subset of fields of YEAR and MONTH, whereas a day-time interval is made up of a contiguous subset of fields consisting of DAY, HOUR, MINUTE, and SECOND. The actual subset of fields that constitute an interval is called the precision of the interval and is specified in the interval qualifier. Because the number of days in a year is calendar-dependent, the year-month interval is NLS-dependent, whereas day-time interval is NLS-independent.

The interval qualifier may also specify the leading field precision, which is the number of digits in the leading or only field, and in case the trailing field is SECOND, it may also specify the fractional seconds precision, which is the number of digits in the fractional part of the SECOND value. If not specified, the default value for leading field precision is 2 digits, and the default value for fractional seconds precision is 6 digits.

INTERVAL YEAR (year_precision) TO MONTH

This data type stores a period of time in years and months, where `year_precision` is the number of digits in the `YEAR` datetime field. The accepted values are 0 through 9. The default is 6.

INTERVAL DAY (day_precision) TO SECOND (fractional_seconds_precision)

This data type stores a period of time in days, hours, minutes, and seconds, where `day_precision` is the maximum number of digits in the `DAY` datetime field (accepted values are 0 through 9; the default is 2), and `fractional_seconds_precision` is the number of digits in the fractional part of the `SECOND` field. The accepted values are 0 through 9. The default is 6.

INTERVAL Fields

INTERVAL Field	Valid Values for Interval
YEAR	Any positive or negative integer
MONTH	00 to 11
DAY	Any positive or negative integer
HOUR	00 to 23
MINUTE	00 to 59
SECOND	00 to 59.9(N) where 9(N) is precision

INTERVAL YEAR TO MONTH can have fields of YEAR and MONTH.

INTERVAL DAY TO SECOND can have fields of DAY, HOUR, MINUTE, and SECOND.

The actual subset of fields that constitute an item of either type of interval is defined by an interval qualifier, and this subset is known as the precision of the item.

Year-month intervals are mutually comparable and assignable only with other year-month intervals, and day-time intervals are mutually comparable and assignable only with other day-time intervals.

INTERVAL YEAR TO MONTH: Example

```
CREATE TABLE warranty
(prod_id number,  warranty_time INTERVAL YEAR(3) TO
MONTH);
INSERT INTO warranty VALUES (123, INTERVAL '8' MONTH);
INSERT INTO warranty VALUES (155, INTERVAL '200'
YEAR(3));
INSERT INTO warranty VALUES (678, '200-11');
SELECT * FROM warranty;
```

	PROD_ID	WARRANTY_TIME
1	123	0-8
2	155	200-0
3	678	200-11

`INTERVAL YEAR TO MONTH` stores a period of time using the `YEAR` and `MONTH` datetime fields. Specify `INTERVAL YEAR TO MONTH` as follows:

`INTERVAL YEAR [(year_precision)] TO MONTH`

where `year_precision` is the number of digits in the `YEAR` datetime field. The default value of `year_precision` is 2.

Restriction: The leading field must be more significant than the trailing field. For example, `INTERVAL '0-1' MONTH TO YEAR` is not valid.

Examples

- `INTERVAL '123-2' YEAR(3) TO MONTH`
  Indicates an interval of 123 years, 2 months
- `INTERVAL '123' YEAR(3)`
  Indicates an interval of 123 years, 0 months
- `INTERVAL '300' MONTH(3)`
  Indicates an interval of 300 months
- `INTERVAL '123' YEAR`
  Returns an error because the default precision is 2, and '123' has 3 18

The Oracle database supports two interval data types: `INTERVAL YEAR TO MONTH` and `INTERVAL DAY TO SECOND`; the column type, PL/SQL argument, variable, and return type must be one of the two. However, for interval literals, the system recognizes other American National Standards Institute (ANSI) interval types such as `INTERVAL '2' YEAR` or `INTERVAL '10' HOUR`. In these cases, each interval is converted to one of the two supported types.

In the example in the slide, a `WARRANTY` table is created, which contains a `warranty_time` column that takes the `INTERVAL YEAR(3) TO MONTH` data type. Different values are inserted into it to indicate years and months for various products. When these rows are retrieved from the table, you see a year value separated from the month value by a (`-`).

INTERVAL DAY TO SECOND Data Type: Example

```
CREATE TABLE lab
( exp_id number, test_time INTERVAL DAY(2) TO SECOND);

INSERT INTO lab VALUES (100012, '90 00:00:00');
INSERT INTO lab VALUES (56098,
  INTERVAL '6 03:30:16' DAY  TO SECOND);
```

```
SELECT * FROM lab;
```

	EXP_ID	TEST_TIME
1	100012	90 0:0:0.0
2	56098	6 3:30:16.0

In the example in the slide, you create the `lab` table with a `test_time` column of the `INTERVAL DAY TO SECOND` data type. You then insert into it the value `'90 00:00:00'` to indicate 90 days and 0 hours, 0 minutes, and 0 seconds, and `INTERVAL '6 03:30:16' DAY TO SECOND` to indicate 6 days, 3 hours, 30 minutes, and 16 seconds. The `SELECT` statement shows how this data is displayed in the database.

Lesson Agenda

- `CURRENT_DATE`, `CURRENT_TIMESTAMP`, and `LOCALTIMESTAMP`
- `INTERVAL` data types
- Using the following functions:
  - `EXTRACT`
  - `TZ_OFFSET`
  - `FROM_TZ`
  - `TO_TIMESTAMP`
  - `TO_YMINTERVAL`
  - `TO_DSINTERVAL`

EXTRACT

- Display all employees who were hired after 2007.

```
SELECT last_name, employee_id, hire_date
  FROM employees
  WHERE EXTRACT(YEAR FROM TO_DATE(hire_date, 'DD-MON-RR')) > 2007
  ORDER BY hire_date;
```

- Display the MONTH component from the HIRE_DATE for those employees whose MANAGER_ID is 100.

```
SELECT last_name, hire_date,
       EXTRACT (MONTH FROM HIRE_DATE)
FROM employees
WHERE manager_id = 100;
```

	LAST_NAME	HIRE_DATE	EXTRACT(MONTHFROMHIRE_DATE)
1	Kochhar	21-SEP-05	9
2	De Haan	13-JAN-01	1
3	Raphaely	07-DEC-02	12
4	Weiss	18-JUL-04	7
5	Fripp	10-APR-05	4
6	Kaufling	01-MAY-03	5

...

The EXTRACT expression extracts and returns the value of a specified datetime field from a datetime or interval value expression. You can extract any of the components mentioned in the following syntax using the EXTRACT function. The syntax of the EXTRACT function is:

```
SELECT EXTRACT( { YEAR | MONTH | DAY | HOUR | MINUTE |
SECOND
  | TIMEZONE_HOUR
  | TIMEZONE_MINUTE
  | TIMEZONE_REGION
  | TIMEZONE_ABBR }
 FROM { expr } )
```

When you extract a TIMEZONE_REGION or TIMEZONE_ABBR (abbreviation), the value returned is a string containing the appropriate time zone name or abbreviation. When you extract any of the other values, the value returned is a date in the Gregorian calendar. When extracting from a datetime with a time zone value, the value returned is in UTC.

In the first example in the slide, the EXTRACT function is used to select all employees who were hired after 2007. In the second example in the slide, the EXTRACT function is used to extract the MONTH from the HIRE_DATE column of the EMPLOYEES table for those employees who report to the manager whose EMPLOYEE_ID is 100.

TZ_OFFSET

Display the time zone offset for the 'US/Eastern', 'Canada/Yukon' and 'Europe/London' time zones:

```
SELECT TZ_OFFSET('US/Eastern'),
       TZ_OFFSET('Canada/Yukon'),
       TZ_OFFSET('Europe/London')
 FROM DUAL;
```

	TZ_OFFSET('US/EASTERN')	TZ_OFFSET('CANADA/YUKON')	TZ_OFFSET('EUROPE/LONDON')
1	-04:00	-07:00	+01:00

ORACLE

The `TZ_OFFSET` function returns the time zone offset corresponding to the value entered. The return value is dependent on the date when the statement is executed. For example, if the `TZ_OFFSET` function returns a value –08:00, this value indicates that the time zone where the command was executed is eight hours behind UTC. You can enter a valid time zone name, a time zone offset from UTC (which simply returns itself), or the keyword `SESSIONTIMEZONE` or `DBTIMEZONE`. The syntax of the `TZ_OFFSET` function is:

```
TZ_OFFSET({ 'time_zone_name'| '{ + | - } hh : mi'
          | SESSIONTIMEZONE
          | DBTMEZONE
          })
```

The Fold Motor Company has its headquarters in Michigan, USA, which is in the US/Eastern time zone. The company president, Mr. Fold, wants to conduct a conference call with the vice president of the Canadian operations and the vice president of European operations, who are in the Canada/Yukon and Europe/London time zones, respectively. Mr. Fold wants to find out the time in each of these places to make sure that his senior management will be available to attend the meeting. His secretary, Mr. Scott, helps by issuing the queries shown in the example and gets the following results:

- The `'US/Eastern'` time zone is four hours behind UTC.
- The `'Canada/Yukon'` time zone is seven hours behind UTC.
- The `'Europe/London'` time zone is one hour ahead of UTC.

For a listing of valid time zone name values, you can query the `V$TIMEZONE_NAMES` dynamic performance view.

```
SELECT * FROM V$TIMEZONE_NAMES;
```

	TZNAME	TZABBREV	CON_ID
1	Africa/Abidjan	LMT	0
2	Africa/Abidjan	GMT	0
3	Africa/Accra	LMT	0
4	Africa/Accra	GMT	0
5	Africa/Accra	GHST	0

...

FROM_TZ

Display the TIMESTAMP value '2000-07-12 08:00:00' as a TIMESTAMP WITH TIME ZONE value for the 'Australia/North' time zone region.

```
SELECT FROM_TZ(TIMESTAMP
       '2000-07-12 08:00:00', 'Australia/North')
FROM DUAL;
```

	FROM_TZ(TIMESTAMP'2000-07-1208:00:00','AUSTRALIA/NORTH')
1	12-JUL-00 08.00.00.000000000 AM AUSTRALIA/NORTH

The FROM_TZ function converts a TIMESTAMP value to a TIMESTAMP WITH TIME ZONE value.

The syntax of the FROM_TZ function is as follows:

```
FROM_TZ(timestamp_value, time_zone_value)
```

where time_zone_value is a character string in the format 'TZH:TZM' or a character expression that returns a string in TZR (time zone region) with an optional TZD format. TZD is an abbreviated time zone string with daylight saving information. TZR represents the time zone region in datetime input strings. Examples are 'Australia/North', 'PST' for US/Pacific standard time, 'PDT' for US/Pacific daylight time, and so on.

The example in the slide converts a TIMESTAMP value to TIMESTAMP WITH TIME ZONE.

Note: To see a listing of valid values for the TZR and TZD format elements, query the V$TIMEZONE_NAMES dynamic performance view.

TO_TIMESTAMP

Display the character string '2007-03-06 11:00:00' as a TIMESTAMP value:

```
SELECT TO_TIMESTAMP ('2007-03-06 11:00:00',
                     'YYYY-MM-DD HH:MI:SS')
FROM DUAL;
```

	TO_TIMESTAMP('2007-03-0611:00:00','YYYY-MM-DDHH:MI:SS')
1	06-MAR-07 11.00.00.000000000 AM

The TO_TIMESTAMP function converts a string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of the TIMESTAMP data type. The syntax of the TO_TIMESTAMP function is:

```
TO_TIMESTAMP(char [, fmt [, 'nlsparam' ] ])
```

The optional fmt specifies the format of char. If you omit fmt, the string must be in the default format of the TIMESTAMP data type. The optional nlsparam specifies the language in which month and day names, and abbreviations, are returned. This argument can have this form:

```
'NLS_DATE_LANGUAGE = language'
```

If you omit nlsparams, this function uses the default date language for your session.

The example in the slide converts a character string to a value of TIMESTAMP.

Note: You use the TO_TIMESTAMP_TZ function to convert a string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of the TIMESTAMP WITH TIME ZONE data type. For more information about this function, see *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

TO_YMINTERVAL

Display a date that is one year and two months after the hire date for the employees working in the department with the DEPARTMENT_ID 20.

```
SELECT hire_date,
       hire_date + TO_YMINTERVAL('01-02') AS
       HIRE_DATE_YMININTERVAL
FROM   employees
WHERE department_id = 20;
```

	HIRE_DATE	HIRE_DATE_YMININTERVAL
1	17-FEB-04	17-APR-05
2	17-AUG-05	17-OCT-06

The TO_YMINTERVAL function converts a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to an INTERVAL YEAR TO MONTH data type. The INTERVAL YEAR TO MONTH data type stores a period of time using the YEAR and MONTH datetime fields. The format of INTERVAL YEAR TO MONTH is as follows:

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

where year_precision is the number of digits in the YEAR datetime field. The default value of year_precision is 2.

The syntax of the TO_YMINTERVAL function is:

```
TO_YMINTERVAL (char)
```

where char is the character string to be converted.

The example in the slide calculates a date that is one year and two months after the hire date for the employees working in the department 20 of the EMPLOYEES table.

TO_DSINTERVAL

Display a date that is 100 days and 10 hours after the hire date for all the employees.

```
SELECT last_name,
 TO_CHAR(hire_date, 'mm-dd-yy:hh:mi:ss') hire_date,
  TO_CHAR(hire_date +
   TO_DSINTERVAL('100 10:00:00'),
     'mm-dd-yy:hh:mi:ss') hiredate2
FROM employees;
```

	LAST_NAME	HIRE_DATE	HIREDATE2
1	King	06-17-03:12:00:00	09-25-03:10:00:00
2	Kochhar	09-21-05:12:00:00	12-30-05:10:00:00
3	De Haan	01-13-01:12:00:00	04-23-01:10:00:00
4	Hunold	01-03-06:12:00:00	04-13-06:10:00:00
5	Ernst	05-21-07:12:00:00	08-29-07:10:00:00
6	Austin	06-25-05:12:00:00	10-03-05:10:00:00
7	Pataballa	02-05-06:12:00:00	05-16-06:10:00:00
8	Lorentz	02-07-07:12:00:00	05-18-07:10:00:00
9	Greenberg	08-17-02:12:00:00	11-25-02:10:00:00
10	Faviet	08-16-02:12:00:00	11-24-02:10:00:00
11	Chen	09-28-05:12:00:00	01-06-06:10:00:00

…

`TO_DSINTERVAL` converts a character string of the `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type to an `INTERVAL DAY TO SECOND` data type.

In the example in the slide, the date 100 days and 10 hours after the hire date is obtained.

Daylight Saving Time (DST)

- Start of Daylight Saving:
  - Time jumps from 01:59:59 AM to 03:00:00 AM.
  - Values from 02:00:00 AM to 02:59:59 AM are not valid.
- End of Daylight Saving:
  - Time jumps from 02:00:00 AM to 01:00:01 AM.
  - Values from 01:00:01 AM to 02:00:00 AM are ambiguous because they are visited twice.

Most western nations advance the clock ahead one hour during the summer months. This period is called daylight saving time. Daylight saving time lasts from the start of Daylight Saving to the end of Daylight Saving in most of the United States, Mexico, and Canada. The nations of the European Union observe daylight saving time, but they call it the summer time period. Europe's summer time period begins a week earlier than its North American counterpart, but ends at the same time.

The Oracle database automatically determines, for any given time zone region, whether daylight saving time is in effect and returns local time values accordingly. The datetime value is sufficient for the Oracle database to determine whether daylight saving time is in effect for a given region in all cases except boundary cases. A boundary case occurs during the period when daylight saving time goes into or out of effect. For example, in the US/Eastern region, when daylight saving time goes into effect, the time changes from 01:59:59 AM to 03:00:00 AM. The one-hour interval between 02:00:00 AM and 02:59:59 AM. does not exist. When daylight saving time goes out of effect, the time changes from 02:00:00 AM back to 01:00:01 AM, and the one-hour interval between 01:00:01 AM and 02:00:00 AM is repeated.

ERROR_ON_OVERLAP_TIME

The `ERROR_ON_OVERLAP_TIME` is a session parameter to notify the system to issue an error when it encounters a datetime that occurs in the overlapped period and no time zone abbreviation was specified to distinguish the period.

For example, daylight saving time ends on October 31, at 02:00:01 AM. The overlapped periods are:

- 10/31/2004 01:00:01 AM to 10/31/2004 02:00:00 AM (EDT)
- 10/31/2004 01:00:01 AM to 10/31/2004 02:00:00 AM (EST)

If you input a datetime string that occurs in one of these two periods, you need to specify the time zone abbreviation (for example, EDT or EST) in the input string for the system to determine the period. Without this time zone abbreviation, the system does the following:

If the `ERROR_ON_OVERLAP_TIME` parameter is `FALSE`, it assumes that the input time is standard time (for example, EST). Otherwise, an error is raised.

Quiz

The `TIME_ZONE` session parameter may be set to:

a. A relative offset
b. Database time zone
c. OS local time zone
d. A named region

Answer: b, c, d

Summary

In this lesson, you should have learned how to:

- Use data types similar to `DATE` that store fractional seconds and track time zones
- Use data types that store the difference between two datetime values
- Use the following datetime functions:
  - `CURRENT_DATE`
  - `CURRENT_TIMESTAMP`
  - `LOCALTIMESTAMP`
  - `DBTIMEZONE`
  - `SESSIONTIMEZONE`
  - `EXTRACT`
  - `TZ_OFFSET`
  - `FROM_TZ`
  - `TO_TIMESTAMP`
  - `TO_YMINTERVAL`
  - `TO_DSINTERVAL`

This lesson addressed some of the datetime functions available in the Oracle database.

Practice 10: Overview

This practice covers using the datetime functions.

In this practice, you display time zone offsets, CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP. You also set time zones and use the EXTRACT function.