



# Oracle Database 19c: SQL Workshop

Instructor Guide - Volume II  
D108644GC10 | D109424

## Authors

Don E Bates  
Shilpa Sharma  
Anthony Skrabak  
Apoorva Srinivas

## Technical Contributors and Reviewers

Nancy Greenberg  
Tulika Das  
Jeremy Smyth  
Purjanti Chang  
Tamal Chatterjee

## Publishers

Sujatha Nagendra  
Pavithran Adka  
Sumesh Koshy

1009112020

**Copyright © 2020, Oracle and/or its affiliates.**

### Disclaimer

This document contains proprietary information and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except where your use constitutes "fair use" under copyright law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

### Restricted Rights Notice

If this documentation is delivered to the United States Government or anyone using the documentation on behalf of the United States Government, the following notice is applicable:

**U.S. GOVERNMENT END USERS:** Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software" or "commercial computer software documentation" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

### Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

### Third-Party Content, Products, and Services Disclaimer

This documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

## Contents

### 1 Introduction

Lesson Objectives	1-2
Lesson Agenda	1-3
Course Objectives	1-4
Icons Used in This Course	1-5
Course Roadmap	1-6
Appendices and Practices Used in the Course	1-13
Lesson Agenda	1-14
Oracle Database 19c: Focus Areas	1-15
Oracle Database 19c	1-16
MySQL: A Modern Database for the Digital Age	1-18
High Scalability with MySQL	1-19
MySQL-Supported Operating Systems	1-20
MySQL Enterprise Edition	1-21
Why MySQL Enterprise Edition?	1-22
Oracle Premier Support for MySQL	1-23
MySQL and Oracle Integration	1-24
Lesson Agenda	1-25
Relational and Object Relational Database Management Systems	1-26
Data Storage on Different Media	1-27
Relational Database Concept	1-28
Definition of a Relational Database	1-29
Data Models	1-30
Entity Relationship Model	1-31
Entity Relationship Modeling Conventions	1-32
Relating Multiple Tables	1-34
Relational Database Terminology	1-35
Lesson Agenda	1-36
Human Resources (HR) Application	1-37
Tables Used in This Course	1-38
Tables Used in the Course	1-39
Lesson Agenda	1-40
Using SQL to Query Your Database	1-41
How SQL Works	1-42
SQL Statements Used in the Course	1-43

Development Environments for SQL in Oracle	1-44
Introduction to Oracle Live SQL	1-45
Development Environments for SQL in MySQL	1-46
Lesson Agenda	1-47
Oracle Database Documentation	1-48
Additional Resources for Oracle	1-49
Oracle University: Oracle SQL Training	1-50
Oracle SQL Certification	1-51
MySQL Websites	1-52
MySQL Community Resources	1-53
Oracle University: MySQL Training	1-54
MySQL Certification	1-55
Summary	1-56
Practice 1: Overview	1-57

## **2 Retrieving Data Using the SQL SELECT Statement**

Course Roadmap	2-2
Objectives	2-3
Lesson Agenda	2-4
HR Application Scenario	2-5
Writing SQL Statements	2-6
Basic SELECT Statement	2-7
Selecting All Columns	2-8
Executing SQL Statements with Oracle SQL Developer and SQL*Plus	2-9
Column Heading Defaults in SQL Developer and SQL*Plus	2-10
Executing SQL Statements in MySQL Workbench	2-11
Executing SQL Statements in mysql Command-line Client	2-12
Selecting Specific Columns	2-13
Selecting from dual with Oracle Database	2-14
Selecting Constant Expressions in MySQL	2-15
Lesson Agenda	2-16
Arithmetic Expressions	2-17
Using Arithmetic Operators	2-18
Operator Precedence	2-19
Defining a Null Value	2-20
Null Values in Arithmetic Expressions	2-21
Lesson Agenda	2-22
Defining a Column Alias	2-23
Using Column Aliases	2-24
Lesson Agenda	2-25
Concatenation Operator in Oracle	2-26

Concatenation Function in MySQL – CONCAT()	2-27
Literal Character Strings	2-28
Using Literal Character Strings in Oracle	2-29
Using Literal Character Strings in MySQL	2-30
Alternative Quote (q) Operator in Oracle	2-31
Including a Single Quotation Mark in a String with an Escape Sequence in MySQL	2-32
Duplicate Rows	2-33
Lesson Agenda	2-34
Displaying Table Structure by Using the DESCRIBE Command	2-35
Displaying Table Structure by Using Oracle SQL Developer	2-36
Displaying Table Structure by Using MySQL Workbench	2-37
Summary	2-38
Practice 2: Overview	2-39

### **3 Restricting and Sorting Data**

Course Roadmap	3-2
Objectives	3-3
Lesson Agenda	3-4
Limiting Rows by Using a Selection	3-5
Limiting Rows That Are Selected	3-6
Using the WHERE Clause	3-7
Character Strings and Dates	3-8
Comparison Operators	3-9
Using Comparison Operators	3-10
Range Conditions Using the BETWEEN Operator	3-11
Using the IN Operator	3-12
Pattern Matching Using the LIKE Operator	3-13
Combining Wildcard Symbols	3-14
Using NULL Conditions	3-15
Defining Conditions Using Logical Operators	3-16
Using the AND Operator	3-17
Using the OR Operator	3-18
Using the NOT Operator	3-19
Lesson Agenda	3-20
Rules of Precedence	3-21
Lesson Agenda	3-23
Using the ORDER BY Clause	3-24
Sorting	3-25
Lesson Agenda	3-27
SQL Row Limiting Clause	3-28

Using SQL Row Limiting Clause in a Query in Oracle 3-29  
SQL Row Limiting Clause: Example in Oracle 3-30  
Using SQL Row Limiting Clause in a Query in MySQL 3-31  
SQL Row Limiting Clause: Example in MySQL 3-32  
Lesson Agenda 3-33  
Substitution Variables in Oracle 3-34  
Using the Single-Ampersand Substitution Variable 3-36  
Character and Date Values with Substitution Variables 3-38  
Specifying Column Names, Expressions, and Text 3-39  
Using the Double-Ampersand Substitution Variable 3-40  
Using the Ampersand Substitution Variable in SQL\*Plus 3-41  
Lesson Agenda 3-42  
Using the DEFINE Command in Oracle 3-43  
Using the VERIFY Command in Oracle 3-44  
Using the SET Statement in MySQL 3-45  
Summary 3-46  
Practice 3: Overview 3-47

#### **4 Using Single-Row Functions to Customize Output**

Course Roadmap 4-2  
Objectives 4-3  
HR Application Scenario 4-4  
Lesson Agenda 4-5  
SQL Functions 4-6  
Two Types of SQL Functions 4-7  
Single-Row Functions 4-8  
Lesson Agenda 4-10  
Character Functions 4-11  
Case-Conversion Functions 4-13  
Using Case-Conversion Functions in WHERE Clauses in Oracle 4-14  
Case-Insensitive Queries in MySQL 4-15  
Character-Manipulation Functions 4-16  
Using Character-Manipulation Functions 4-17  
Lesson Agenda 4-18  
Nesting Functions 4-19  
Nesting Functions: Example 4-20  
Lesson Agenda 4-21  
Numeric Functions 4-22  
Using the ROUND Function 4-23  
Using the TRUNC Function in Oracle 4-24  
Using the TRUNCATE Function in MySQL 4-25

Using the MOD Function	4-26
Lesson Agenda	4-27
Working with Dates in Oracle Databases	4-28
RR Date Format in Oracle	4-29
Using the SYSDATE Function in Oracle	4-30
Using the CURRENT_DATE and CURRENT_TIMESTAMP Functions in Oracle	4-31
Arithmetic with Dates in Oracle	4-32
Using Arithmetic Operators with Dates in Oracle	4-33
Lesson Agenda	4-34
Working with Dates in MySQL Databases	4-35
Displaying the Current Date in MySQL	4-36
Lesson Agenda	4-37
Date-Manipulation Functions in Oracle	4-38
Using Date Functions in Oracle	4-39
Using ROUND and TRUNC Functions with Dates in Oracle	4-40
Date-Manipulation Functions in MySQL	4-41
Using Date Functions in MySQL	4-42
Extracting the Month or Year Portion of Dates in MySQL	4-43
Summary	4-44
Practice 4: Overview	4-45
<b>5 Using Conversion Functions and Conditional Expressions</b>	
Course Roadmap	5-2
Objectives	5-3
Lesson Agenda	5-4
Conversion Functions	5-5
Implicit Data Type Conversion of Strings to Numbers	5-6
Implicit Data Type Conversion of Numbers to Strings	5-7
Lesson Agenda	5-8
Using the TO_CHAR Function with Dates	5-9
Elements of the Date Format Model	5-10
Using the TO_CHAR Function with Dates	5-13
Using the TO_CHAR Function with Numbers	5-14
Using the TO_NUMBER and TO_DATE Functions	5-17
Using TO_CHAR and TO_DATE Functions with the RR Date Format	5-19
Lesson Agenda	5-20
Using the CAST() function in Oracle	5-21
Explicit Data Type Conversion of Strings to Numbers in MySQL	5-22
Explicit Data Type Conversion of Numbers to Strings in MySQL	5-23
Lesson Agenda	5-24

General Functions	5-25
NVL Function (Oracle) and IFNULL() Function (MySQL)	5-26
Using the NVL Function in Oracle	5-27
Using the NVL2 Function in Oracle	5-28
Using the IFNULL Function in MySQL	5-29
Using the NULLIF Function	5-30
Using the COALESCE Function	5-31
Lesson Agenda	5-33
Conditional Expressions	5-34
CASE Expression	5-35
Using the CASE Expression	5-36
Searched CASE Expression	5-37
DECODE Function in Oracle	5-38
Using the DECODE Function	5-39
Lesson Agenda	5-41
JSON_QUERY Function	5-42
JSON_TABLE Function	5-43
JSON_VALUE Function	5-44
Summary	5-45
Practice 5: Overview	5-46

## 6 Reporting Aggregated Data Using the Group Functions

Course Roadmap	6-2
Objectives	6-3
Lesson Agenda	6-4
Group Functions	6-5
Types of Group Functions	6-6
Group Functions: Syntax	6-7
Using the AVG and SUM Functions	6-8
Using the MIN and MAX Functions	6-9
Using the COUNT Function	6-10
Using the DISTINCT Keyword	6-11
Group Functions and Null Values in Oracle	6-12
Group Functions and Null Values in MySQL	6-13
Lesson Agenda	6-14
Creating Groups of Data	6-15
Creating Groups of Data: GROUP BY Clause Syntax	6-16
Using the GROUP BY Clause	6-17
Grouping by More Than One Column	6-19
Using the GROUP BY Clause on Multiple Columns	6-20
Illegal Queries Using Group Functions	6-21

Illegal Queries Using Group Functions in a WHERE Clause	6-22
Restricting Group Results	6-23
Restricting Group Results with the HAVING Clause	6-24
Using the HAVING Clause	6-25
Lesson Agenda	6-27
Nesting Group Functions in Oracle	6-28
Summary	6-29
Practice 6: Overview	6-30

## **7 Displaying Data from Multiple Tables Using Joins**

Course Roadmap	7-2
Objectives	7-3
Lesson Agenda	7-4
Why Join?	7-5
Obtaining Data from Multiple Tables	7-6
Types of Joins	7-7
Joining Tables Using SQL Syntax	7-8
Lesson Agenda	7-9
Creating Natural Joins	7-10
Retrieving Records with Natural Joins	7-11
Creating Joins with the USING Clause	7-12
Joining Column Names	7-13
Retrieving Records with the USING Clause	7-14
Qualifying Ambiguous Column Names	7-15
Using Table Aliases with the USING Clause in Oracle	7-16
Creating Joins with the ON Clause	7-17
Retrieving Records with the ON Clause	7-18
Creating Three-Way Joins	7-19
Applying Additional Conditions to a Join	7-20
Lesson Agenda	7-21
Joining a Table to Itself	7-22
Self-Joins Using the ON Clause	7-23
Lesson Agenda	7-24
Nonequijoins	7-25
Retrieving Records with Nonequijoins	7-26
Lesson Agenda	7-27
Returning Records with No Direct Match Using OUTER Joins	7-28
INNER Versus OUTER Joins	7-29
LEFT OUTER JOIN	7-30
RIGHT OUTER JOIN	7-31
FULL OUTER JOIN in Oracle	7-32

Lesson Agenda	7-33
Cartesian Products	7-34
Generating a Cartesian Product	7-35
Creating Cross Joins	7-36
Summary	7-37
Practice 7: Overview	7-38

## **8 Using Subqueries to Solve Queries**

Course Roadmap	8-2
Objectives	8-3
Lesson Agenda	8-4
Using a Subquery to Solve a Problem	8-5
Subquery Syntax	8-6
Using a Subquery	8-7
Rules and Guidelines for Using Subqueries	8-8
Types of Subqueries	8-9
Lesson Agenda	8-10
Single-Row Subqueries	8-11
Executing Single-Row Subqueries	8-12
Using Group Functions in a Subquery	8-13
HAVING Clause with Subqueries	8-14
What Is Wrong with This Statement?	8-15
No Rows Returned by the Inner Query	8-16
Lesson Agenda	8-17
Multiple-Row Subqueries	8-18
Using the ANY Operator in Multiple-Row Subqueries	8-19
Using the ALL Operator in Multiple-Row Subqueries	8-20
Multiple-Column Subqueries	8-21
Multiple-Column Subquery: Example	8-22
Lesson Agenda	8-23
Null Values in a Subquery	8-24
Summary	8-25
Practice 8: Overview	8-26

## **9 Using Set Operators**

Course Roadmap	9-2
Objectives	9-3
Lesson Agenda	9-4
Set Operators	9-5
Set Operator Rules	9-6
Oracle Server and Set Operators	9-7

Lesson Agenda	9-8
Tables Used in This Lesson	9-9
Lesson Agenda	9-13
UNION Operator	9-14
Using the UNION Operator	9-15
UNION ALL Operator	9-16
Using the UNION ALL Operator	9-17
Lesson Agenda	9-18
Matching the SELECT Statement: Example in Oracle	9-26
Matching SELECT Statements in MySQL	9-27
Matching the SELECT Statement: Example in MySQL	9-28
Lesson Agenda	9-29
Using the ORDER BY Clause with UNION in MySQL	9-32
Using the ORDER BY Clause with UNION: Example in MySQL	9-33
Summary	9-34
Practice 9: Overview	9-35

## **10a Managing Tables Using DML Statements in Oracle**

Course Roadmap	10a-2
Objectives	10a-3
HR Application Scenario	10a-4
Lesson Agenda	10a-5
Data Manipulation Language	10a-6
Adding a New Row to a Table	10a-7
INSERT Statement Syntax	10a-8
Inserting New Rows	10a-9
Inserting Rows with Null Values	10a-10
Inserting Special Values	10a-11
Inserting Specific Date and Time Values	10a-12
Creating a Script	10a-13
Copying Rows from Another Table	10a-14
Lesson Agenda	10a-15
Changing Data in a Table	10a-16
UPDATE Statement Syntax	10a-17
Updating Rows in a Table	10a-18
Updating Two Columns with a Subquery	10a-19
Updating Rows Based on Another Table	10a-20
Lesson Agenda	10a-21
Removing a Row from a Table	10a-22
DELETE Statement	10a-23
Deleting Rows from a Table	10a-24

Deleting Rows Based on Another Table	10a-25
TRUNCATE Statement	10a-26
Lesson Agenda	10a-27
Database Transactions	10a-28
Database Transactions: Start and End	10a-29
Advantages of the COMMIT and ROLLBACK Statements	10a-30
Explicit Transaction Control Statements	10a-31
Rolling Back Changes to a Marker	10a-32
Implicit Transaction Processing	10a-33
State of Data Before COMMIT or ROLLBACK	10a-34
State of Data After COMMIT	10a-35
Committing Data	10a-36
State of Data After ROLLBACK	10a-37
State of Data After ROLLBACK: Example	10a-38
Statement-Level Rollback	10a-39
Lesson Agenda	10a-40
Read Consistency	10a-41
Implementing Read Consistency	10a-42
Lesson Agenda	10a-43
FOR UPDATE Clause in a SELECT Statement	10a-44
FOR UPDATE Clause: Examples	10a-45
LOCK TABLE Statement	10a-46
Summary	10a-47
Practice 10a: Overview	10a-48

## **10b Managing Tables Using DML Statements in MySQL**

Course Roadmap	10b-2
Objectives	10b-3
HR Application Scenario	10b-4
Lesson Agenda	10b-5
Data Manipulation Language	10b-6
Adding a New Row to a Table	10b-7
INSERT Statement Syntax	10b-8
Inserting New Rows: Listing Column Names	10b-9
Inserting New Rows: Omitting Column Names	10b-10
Inserting Rows with Null Values	10b-11
Inserting Special Values in MySQL	10b-12
Inserting Specific Date and Time Values in MySQL	10b-13
Inserting and Reformatting Specific Date and Time Values in MySQL	10b-14
Copying Rows from Another Table	10b-15
Lesson Agenda	10b-16

Changing Data in a Table	10b-17
UPDATE Statement Syntax	10b-18
Updating Rows in a Table	10b-19
Updating Rows Based on Another Table	10b-20
Quiz	10b-21
Lesson Agenda	10b-22
Removing a Row from a Table	10b-23
DELETE Statement	10b-24
Deleting Rows from a Table	10b-25
Deleting Rows Based on Another Table	10b-26
TRUNCATE Statement	10b-27
Lesson Agenda	10b-28
Multiple-statement Transactions	10b-29
Transaction Diagram	10b-30
AUTOCOMMIT and Transaction Control Statements	10b-31
Committing Data in a Transaction	10b-32
Rolling Back Changes	10b-33
Rolling Back Changes to a Marker	10b-34
Lesson Agenda	10b-35
Consistent Reads	10b-36
Lesson Agenda	10b-37
FOR UPDATE Clause in a SELECT Statement	10b-38
FOR UPDATE Clause: Examples	10b-39
Summary	10b-40
Practice 10b: Overview	10b-41

## **11a Introduction to Data Definition Language in Oracle**

Course Roadmap	11a-2
Objectives	11a-3
HR Application Scenario	11a-4
Lesson Agenda	11a-5
Database Objects	11a-6
Naming Rules for Tables and Columns	11a-7
Lesson Agenda	11a-8
CREATE TABLE Statement	11a-9
Creating Tables	11a-10
Lesson Agenda	11a-11
Data Types	11a-12
Datetime Data Types	11a-14
DEFAULT Option	11a-15
Lesson Agenda	11a-16

Including Constraints	11a-17
Constraint Guidelines	11a-18
Defining Constraints	11a-19
Defining Constraints: Example	11a-20
NOT NULL Constraint	11a-21
UNIQUE Constraint	11a-22
PRIMARY KEY Constraint	11a-24
FOREIGN KEY Constraint	11a-25
FOREIGN KEY Constraint: Keywords	11a-27
CHECK Constraint	11a-28
CREATE TABLE: Example	11a-29
Violating Constraints	11a-30
Lesson Agenda	11a-32
Creating a Table Using a Subquery	11a-33
Lesson Agenda	11a-35
ALTER TABLE Statement	11a-36
Adding a Column	11a-38
Modifying a Column	11a-39
Dropping a Column	11a-40
SET UNUSED Option	11a-41
Read-Only Tables	11a-43
Lesson Agenda	11a-44
Dropping a Table	11a-45
Summary	11a-46
Practice 11a: Overview	11a-47

## **11b Introduction to Data Definition Language in MySQL**

Course Roadmap	11b-2
Objectives	11b-3
HR Application Scenario	11b-4
Lesson Agenda	11b-5
Creating a Database: Syntax	11b-6
MySQL Naming Conventions	11b-7
Lesson Agenda	11b-8
CREATE TABLE Statement	11b-9
Lesson Agenda	11b-10
Data Types: Overview	11b-11
Numeric Data Types	11b-12
Date and Time Data Types	11b-13
String Data Types	11b-14
Lesson Agenda	11b-15

Indexes, Keys, and Constraints	11b-16
Table Indexes	11b-17
Primary Keys	11b-18
Unique Key Constraints	11b-19
Foreign Key Constraints	11b-20
Foreign Key Constraint: Example Tables	11b-21
FOREIGN KEY Constraint: Example Statement	11b-22
FOREIGN KEY Constraint: Referential Actions	11b-23
Secondary Indexes	11b-24
Lesson Agenda	11b-25
Column Options	11b-26
Lesson Agenda	11b-27
Creating a Table Using a Subquery	11b-28
Creating a Table Using a Subquery: Example	11b-29
Lesson Agenda	11b-30
ALTER TABLE Statement	11b-31
ALTER TABLE Statement: Add, Modify, or Drop Columns	11b-32
Adding a Column	11b-33
Modifying a Column	11b-34
Dropping a Column	11b-35
ALTER TABLE Statement: Add an Index or Constraint	11b-36
ALTER TABLE to Add a Constraint or Index: Example	11b-37
Creating Indexes by Using the CREATE INDEX Statement	11b-38
Viewing Index Definitions by Using the SHOW INDEX Statement	11b-39
Showing How a Table Was Created with the SHOW CREATE	
TABLE Statement	11b-40
Lesson Agenda	11b-41
Dropping a Table	11b-42
Summary	11b-43
Practice 11b: Overview	11b-44

## 12 Introduction to Data Dictionary Views

Course Roadmap	12-2
Objectives	12-3
Lesson Agenda	12-4
Why Data Dictionary?	12-5
Data Dictionary	12-6
Data Dictionary Structure	12-7
How to Use Dictionary Views	12-9
USER_OBJECTS and ALL_OBJECTS Views	12-10
USER_OBJECTS View	12-11

Lesson Agenda 12-12  
Table Information 12-13  
Column Information 12-14  
Constraint Information 12-16  
USER\_CONSTRAINTS: Example 12-17  
Querying USER\_CONS\_COLUMNS 12-18  
Lesson Agenda 12-19  
Adding Comments to a Table 12-20  
Summary 12-21  
Practice 12: Overview 12-22

### **13 Creating Sequences, Synonyms, and Indexes**

Course Roadmap 13-2  
Objectives 13-3  
Lesson Agenda 13-4  
E-Commerce Scenario 13-5  
Database Objects 13-6  
Referencing Another User's Tables 13-7  
Sequences 13-8  
CREATE SEQUENCE Statement: Syntax 13-9  
Creating a Sequence 13-11  
NEXTVAL and CURRVAL Pseudocolumns 13-12  
Using a Sequence 13-14  
SQL Column Defaulting Using a Sequence 13-15  
Caching Sequence Values 13-16  
Modifying a Sequence 13-17  
Guidelines for Modifying a Sequence 13-18  
Sequence Information 13-19  
Lesson Agenda 13-20  
Synonyms 13-21  
Creating a Synonym for an Object 13-22  
Creating and Removing Synonyms 13-23  
Synonym Information 13-24  
Lesson Agenda 13-25  
Indexes 13-26  
How Are Indexes Created? 13-27  
Creating an Index 13-28  
CREATE INDEX with the CREATE TABLE Statement 13-29  
Function-Based Indexes 13-31  
Creating Multiple Indexes on the Same Set of Columns 13-32  
Creating Multiple Indexes on the Same Set of Columns: Example 13-33

Index Information	13-34
USER_INDEXES: Examples	13-35
Querying USER_IND_COLUMNS	13-36
Removing an Index	13-37
Summary	13-38
Practice 13: Overview	13-39

## **14 Creating Views**

Course Roadmap	14-2
Objectives	14-3
Lesson Agenda	14-4
Why Views?	14-5
Database Objects	14-6
What Is a View?	14-7
Advantages of Views	14-8
Simple Views and Complex Views	14-9
Lesson Agenda	14-10
Creating a View	14-11
Retrieving Data from a View	14-14
Modifying a View	14-15
Creating a Complex View	14-16
View Information	14-17
Lesson Agenda	14-18
Rules for Performing DML Operations on a View	14-19
Rules for Performing Modify Operations on a View	14-20
Rules for Performing Insert Operations Through a View	14-21
Using the WITH CHECK OPTION Clause	14-22
Denying DML Operations	14-23
Lesson Agenda	14-25
Removing a View	14-26
Summary	14-27
Practice 14: Overview	14-28

## **15 Managing Schema Objects**

Course Roadmap	15-2
Objectives	15-3
Lesson Agenda	15-4
Adding a Constraint Syntax	15-5
Adding a Constraint	15-6
Dropping a Constraint	15-7
Dropping a Constraint ONLINE	15-8

ON DELETE Clause	15-9
Cascading Constraints	15-10
Renaming Table Columns and Constraints	15-12
Disabling Constraints	15-13
Enabling Constraints	15-14
Constraint States	15-15
Deferring Constraints	15-16
Difference Between INITIALLY DEFERRED and INITIALLY IMMEDIATE	15-17
DROP TABLE ... PURGE	15-19
Lesson Agenda	15-20
Using Temporary Tables	15-21
Temporary Table	15-22
Temporary Table Characteristics	15-23
Creating a Global Temporary Table	15-24
Creating a Private Temporary Table	15-25
Lesson Agenda	15-26
External Tables	15-27
Creating a Directory for the External Table	15-28
Creating an External Table	15-30
Creating an External Table by Using ORACLE_LOADER	15-32
Querying External Tables	15-33
Creating an External Table by Using ORACLE_DATAPUMP: Example	15-34
Summary	15-35
Practice 15: Overview	15-36

## **16 Retrieving Data by Using Subqueries**

Course Roadmap	16-2
Objectives	16-3
Lesson Agenda	16-4
Retrieving Data by Using a Subquery as a Source	16-5
Lesson Agenda	16-7
Multiple-Column Subqueries	16-8
Column Comparisons	16-9
Pairwise Comparison Subquery	16-10
Nonpairwise Comparison Subquery	16-11
Lesson Agenda	16-12
Scalar Subquery Expressions	16-13
Scalar Subqueries: Examples	16-14
Lesson Agenda	16-15
Correlated Subqueries	16-16
Using Correlated Subqueries: Example 1	16-18

Using Correlated Subqueries: Example 2	16-19
Lesson Agenda	16-20
Using the EXISTS Operator	16-21
Find All Departments That Do Not Have Any Employees	16-23
Lesson Agenda	16-24
WITH Clause	16-25
WITH Clause: Example	16-26
Recursive WITH Clause	16-27
Recursive WITH Clause: Example	16-28
Summary	16-29
Practice 16: Overview	16-30

## **17 Manipulating Data by Using Subqueries**

Course Roadmap	17-2
Objectives	17-3
Lesson Agenda	17-4
Using Subqueries to Manipulate Data	17-5
Lesson Agenda	17-6
Inserting by Using a Subquery as a Target	17-7
Lesson Agenda	17-9
Using the WITH CHECK OPTION Keyword on DML Statements	17-10
Lesson Agenda	17-12
Correlated UPDATE	17-13
Using Correlated UPDATE	17-14
Correlated DELETE	17-16
Using Correlated DELETE	17-17
Summary	17-18
Practice 17: Overview	17-19

## **18 Controlling User Access**

Course Roadmap	18-2
Objectives	18-3
Lesson Agenda	18-4
Controlling User Access	18-5
Privileges	18-6
System Privileges	18-7
Creating Users	18-8
User System Privileges	18-9
Granting System Privileges	18-10
Lesson Agenda	18-11
What Is a Role?	18-12

Creating and Granting Privileges to a Role	18-13
Changing Your Password	18-14
Lesson Agenda	18-15
Object Privileges	18-16
Granting Object Privileges	18-18
Passing On Your Privileges	18-19
Confirming Granted Privileges	18-20
Lesson Agenda	18-21
Revoking Object Privileges	18-22
Summary	18-24
Practice 18: Overview	18-25

## **19 Manipulating Data Using Advanced Queries**

Course Roadmap	19-2
Objectives	19-3
Lesson Agenda	19-4
Explicit Default Feature: Overview	19-5
Using Explicit Default Values	19-6
Lesson Agenda	19-7
E-Commerce Scenario	19-8
Multitable INSERT Statements: Overview	19-9
Types of Multitable INSERT Statements	19-11
Multitable INSERT Statements	19-12
Unconditional INSERT ALL	19-14
Conditional INSERT ALL: Example	19-15
Conditional INSERT ALL	19-16
Conditional INSERT FIRST: Example	19-18
Conditional INSERT FIRST	19-19
Pivoting INSERT	19-20
Lesson Agenda	19-23
MERGE Statement	19-24
MERGE Statement Syntax	19-25
Merging Rows: Example	19-26
Lesson Agenda	19-29
FLASHBACK TABLE Statement	19-30
Using the FLASHBACK TABLE Statement	19-32
Lesson Agenda	19-33
Tracking Changes in Data	19-34
Flashback Query: Example	19-35

Flashback Version Query: Example 19-36  
VERSIONS BETWEEN Clause 19-37  
Summary 19-38  
Practice 19: Overview 19-39

## 20 Managing Data in Different Time Zones

Course Roadmap 20-2  
Objectives 20-3  
Lesson Agenda 20-4  
E-Commerce Scenario 20-5  
Time Zones 20-6  
TIME\_ZONE Session Parameter 20-7  
CURRENT\_DATE, CURRENT\_TIMESTAMP, and LOCALTIMESTAMP 20-8  
Comparing Date and Time in a Session's Time Zone 20-9  
DBTIMEZONE and SESSIONTIMEZONE 20-11  
TIMESTAMP Data Types 20-12  
TIMESTAMP Fields 20-13  
Difference Between DATE and TIMESTAMP 20-14  
Comparing TIMESTAMP Data Types 20-15  
Lesson Agenda 20-16  
INTERVAL Data Types 20-17  
INTERVAL Fields 20-18  
INTERVAL YEAR TO MONTH: Example 20-19  
INTERVAL DAY TO SECOND Data Type: Example 20-21  
Lesson Agenda 20-22  
EXTRACT 20-23  
TZ\_OFFSET 20-24  
FROM\_TZ 20-26  
TO\_TIMESTAMP 20-27  
TO\_YMINTERVAL 20-28  
TO\_DSINTERVAL 20-29  
Daylight Saving Time (DST) 20-30  
Summary 20-32  
Practice 20: Overview 20-33

## 21 Conclusion

Course Goals 21-2  
Oracle University: Oracle SQL Training 21-3  
Oracle University: MySQL Training 21-4  
Oracle SQL References 21-5  
MySQL Websites 21-6

Your Evaluation 21-7

Thank You 21-8

Q&A Session 21-9

## 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 17.4.1 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-18

Saving SQL Scripts B-19

Executing Saved Script Files: Method 1 B-20

Executing Saved Script Files: Method 2 B-21

Formatting the SQL Code B-22

Using Snippets B-23

Using Snippets: Example B-24

Using the Recycle Bin B-25

Debugging Procedures and Functions B-26

Database Reporting B-27

Creating a User-Defined Report B-28

External Tools B-29

Setting Preferences B-30

Data Modeler in SQL Developer B-31

Summary B-32

### C Using SQL\*Plus

Objectives C-2

SQL and SQL\*Plus Interaction C-3

SQL Statements Versus SQL\*Plus Commands C-4

SQL\*Plus: Overview 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 and 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

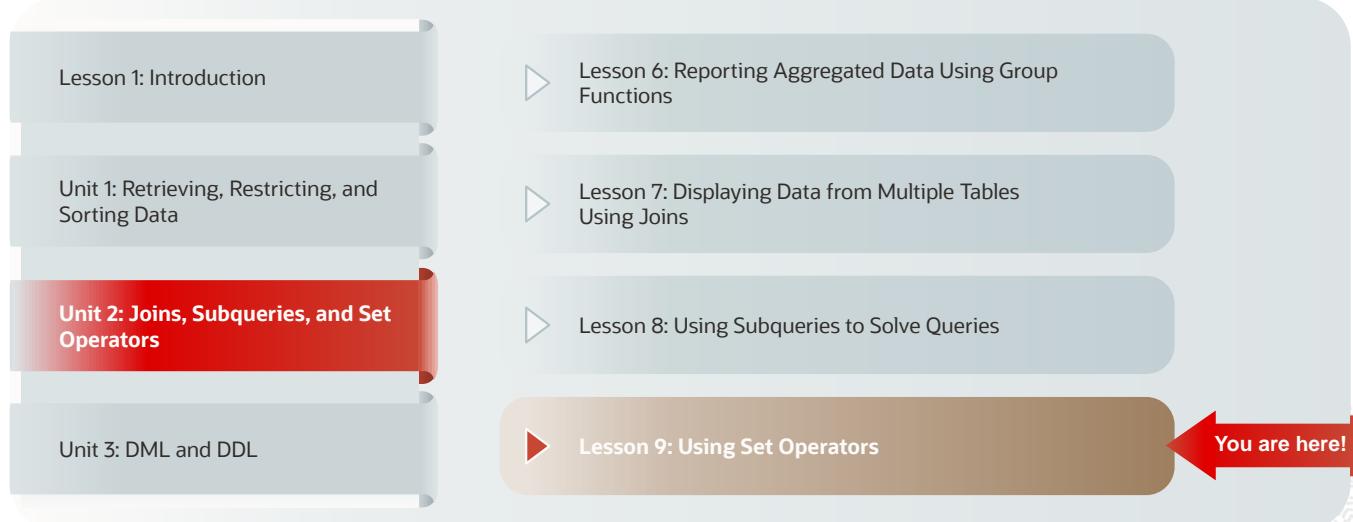
## **I Regular Expression Support**

- 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

# Using Set Operators

# Course Roadmap



2

In Unit 2, you will learn to use:

- SQL statements to query and display data from multiple tables using Joins
- Subqueries when the condition is unknown
- Group functions to aggregate data
- Set operators

# Objectives

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

- Describe set operators
- Use a set operator to combine multiple queries into a single query
- Control the order of rows returned

0

3

In this lesson, you learn how to write queries by using set operators.



# Lesson Agenda

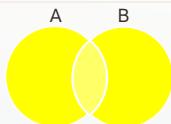
- Set operators: Types and guidelines
- Tables used in this lesson
- UNION and UNION ALL operator
- INTERSECT operator
- MINUS operator
- Matching SELECT statements
- Using the ORDER BY clause in set operations



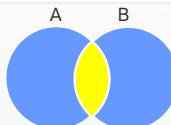
0

For Instructor Use Only.  
This document should not be distributed.

# Set Operators



UNION/UNION ALL



INTERSECT



MINUS

5

0

Set operators combine the results of two or more component queries into one result. Queries containing set operators are called *compound queries*.

Operator	Returns
UNION	Rows from both queries after eliminating duplicates
UNION ALL	Rows from both queries, including all duplicates
INTERSECT	Rows that are common to both queries
MINUS	Rows in the first query that are not present in the second query

All set operators have equal precedence. If a SQL statement contains multiple set operators, the Oracle server evaluates them from left (top) to right (bottom), if no parentheses explicitly specify another order. You should use parentheses to specify the order of evaluation explicitly in queries that use the INTERSECT operator with other set operators.

# Set Operator Rules

- The expressions in the `SELECT` lists must match in number.
- The data type of each column in the subsequent query must match the data type of its corresponding column in the first query.
- Parentheses can be used to alter the sequence of execution.
- The `ORDER BY` clause can appear only at the very end of the statement.



0

6

- The expressions in the `SELECT` lists of the queries must match in number and data type. When you use the `UNION`, `UNION ALL`, `INTERSECT`, and `MINUS` operators, ensure that the `SELECT` lists have the same number and data type of columns. The data type sometimes may not be exactly the same. In such cases, the column in the second query must be in the same data type group (such as numeric or character) as the corresponding column in the first query.
- You can use the set operators in subqueries.
- You should use parentheses to specify the order of evaluation in queries that use the `INTERSECT` operator with other set operators. According to SQL standards, the `INTERSECT` operator has greater precedence than the other set operators.

# Oracle Server and Set Operators

- Duplicate rows are automatically eliminated except in UNION ALL.
- Column names from the first query appear in the result.
- The output is sorted in ascending order by default, except in UNION ALL.

7



O

The `SELECT` lists of a compound query must match the corresponding expressions in number and data type. If component queries select character data, the data type of the return values is determined as follows:

- If both queries select values of `CHAR` data type, of equal length, the returned values have the `CHAR` data type of that length. If the queries select values of `CHAR` with different lengths, the returned value is `VARCHAR2` with the length of the larger `CHAR` value.
- If either or both of the queries select values of `VARCHAR2` data type, the returned values have the `VARCHAR2` data type.
- If component queries select numeric data, the data type of the return values is determined by numeric precedence.
- If all queries select values of the `NUMBER` type, the returned values have the `NUMBER` data type.
- In queries using set operators, the Oracle server does not perform implicit conversion across data type groups. Therefore, if the corresponding expressions of component queries resolve to both character data and numeric data, the Oracle server returns an error.

# Lesson Agenda

- Set operators: Types and guidelines
- Tables used in this lesson
- UNION and UNION ALL operator
- INTERSECT operator
- MINUS operator
- Matching SELECT statements
- Using the ORDER BY clause in set operations

8

0



For Instructor Use Only.  
This document should not be distributed.

# Tables Used in This Lesson

The tables used in this lesson are:

- `employees`: Provides details about all current employees
- `retired_employees`: Provides details about all past employees



Two tables are used in this lesson: `employees` and `retired_employees`.

You are already familiar with the `employees` table that stores employee details, such as a unique identification number, email address, job identification (such as `ST_CLERK`, `SA_REP`, and so on), salary, manager, and so on.

`retired_employees` stores the details of the employees who have left the company.

The structure of and data from the `employees` and `retired_employees` tables are shown on the following pages.

```
DESCRIBE employees
```

DESCRIBE employees		
Name	Null	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

For Instructor Use Only.  
This document should not be distributed.

```
SELECT employee_id, last_name, job_id, hire_date, department_id
FROM employees;
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	HIRE_DATE	DEPARTMENT_ID
1	100	King	AD_PRES	17-JUN-11	90
2	101	Kochhar	AD_VP	21-SEP-09	90
3	102	De Haan	AD_VP	13-JAN-09	90
4	103	Hunold	IT_PROG	03-JAN-14	60
5	104	Ernst	IT_PROG	21-MAY-15	60
6	107	Lorentz	IT_PROG	07-FEB-15	60
7	124	Mourgos	ST_MAN	16-NOV-15	50
8	141	Rajs	ST_CLERK	17-OCT-11	50
9	142	Davies	ST_CLERK	29-JAN-13	50
10	143	Matos	ST_CLERK	15-MAR-14	50
11	144	Vargas	ST_CLERK	09-JUL-14	50
12	149	Zlotkey	SA_MAN	29-JAN-16	80
13	174	Abel	SA_REP	11-MAY-12	80
14	176	Taylor	SA_REP	24-MAR-14	80
15	178	Grant	SA_REP	24-MAY-15	(null)
16	200	Whalen	AD_ASST	17-SEP-11	10
17	201	Hartstein	MK_MAN	17-FEB-12	20
18	202	Fay	MK_REP	17-AUG-13	20
19	205	Higgins	AC_MGR	07-JUN-10	110
20	206	Gietz	AC_ACCOUNT	07-JUN-10	110

```
DESCRIBE retired_employees
```

Name	Null	Type
EMPLOYEE_ID		NUMBER(7)
FIRST_NAME		VARCHAR2(20)
LAST_NAME		VARCHAR2(20)
EMAIL		VARCHAR2(25)
RETIRED_DATE		DATE
JOB_ID		VARCHAR2(20)
SALARY		NUMBER(8,2)
MANAGER_ID		NUMBER(4)
DEPARTMENT_ID		NUMBER(6)

```
SELECT * FROM retired_employees;
```

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	RETIRE_DATE	JOB_ID	SALARY	MANAGER_ID	DEPARTMENT_ID
1	301	Rick	Dayle	RDAYLE	18-MAR-10	AD_PRES	8000	124	90
2	302	Meena	Rac	MRAC	21-SEP-11	AD_VP	11000	149	90
3	303	Mex	Haan	MHAAN	13-JAN-10	AD_VP	9500	149	80
4	304	Alexandera	Runold	ARUNOLD	03-JAN-11	IT_PROG	7500	124	60
5	305	Bruk	Ernst	BERNST	21-MAY-10	IT_PROG	6000	149	60
6	306	Dravid	Aust	DAUST	25-JUN-09	IT_PROG	4800	124	60
7	307	Raj	Patil	RPATIL	05-FEB-12	IT_PROG	4800	201	60
8	308	Rahul	Bose	RBOSE	17-AUG-12	FI_MGR	12008	124	100
9	309	Dany	Fav	DFAV	16-AUG-11	FI_ACCOUNT	9000	101	100
10	310	James	Ken	JKHEN	28-SEP-10	FI_ACCOUNT	8200	101	90
11	311	Shana	Garg	SGARG	30-SEP-10	FI_ACCOUNT	7700	201	100
12	312	Peter	Jois	PJOIS	07-JUN-14	FI_ACCOUNT	7800	124	100
13	313	Lui	Pops	LPOPS	07-DEC-10	FI_ACCOUNT	6900	201	100
14	314	Del	Raph	DRAPH	07-DEC-12	PU_MAN	11000	101	30
15	315	Alex	Khurl	AKHURL	18-MAY-11	PU_CLERK	3100	149	30

For Instructor Use Only  
This document should not be distributed.

# Lesson Agenda

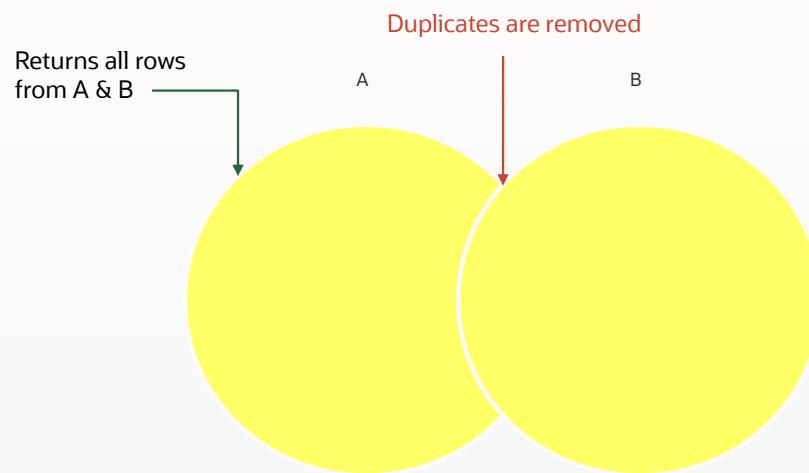
- Set operators: Types and guidelines
- Tables used in this lesson
- UNION and UNION ALL operator
- INTERSECT operator
- MINUS operator
- Matching SELECT statements
- Using the ORDER BY clause in set operations



0

For Instructor Use Only.  
This document should not be distributed.

# UNION Operator



The UNION operator returns rows from both queries after eliminating duplications.

14

O

The UNION operator returns all rows that are selected by either query. Use the UNION operator to return all rows from multiple tables and eliminate any duplicate rows.

## Guidelines

- The number of columns being selected must be the same.
- The data types of the columns being selected must be in the same data type group (such as numeric or character).
- The names of the columns need not be identical.
- UNION operates over all of the columns being selected.
- NULL values are not ignored during duplicate checking.
- By default, the output is sorted in ascending order of the columns of the SELECT clause.

# Using the UNION Operator

Display the job details of all the current and retired employees. Display each job only once.

```
SELECT job_id  
FROM employees  
UNION  
SELECT job_id  
FROM retired_employees;
```



#	job_id
1	AC_ACCOUNT
2	AC_MGR
3	AD_ASST
4	AD_PRES
5	AD_VP
6	FI_ACCOUNT
7	FI_MGR
8	IT_PROG
9	MK_MAN
10	MK_REP
11	PU_CLERK
12	PU_MAN
13	SA_MAN
14	SA_REP
15	ST_CLERK
16	ST_MAN



#	job_id
1	AC_ACCOUNT
2	AC_MGR
3	AD_ASST
4	AD_PRES
5	AD_VP
6	IT_PROG
7	MK_MAN
8	MK_REP
9	SA_MAN
10	SA_REP
11	ST_CLERK
12	ST_MAN
13	FI_MGR
14	FI_ACCOUNT
15	PU_MAN
16	PU_CLERK

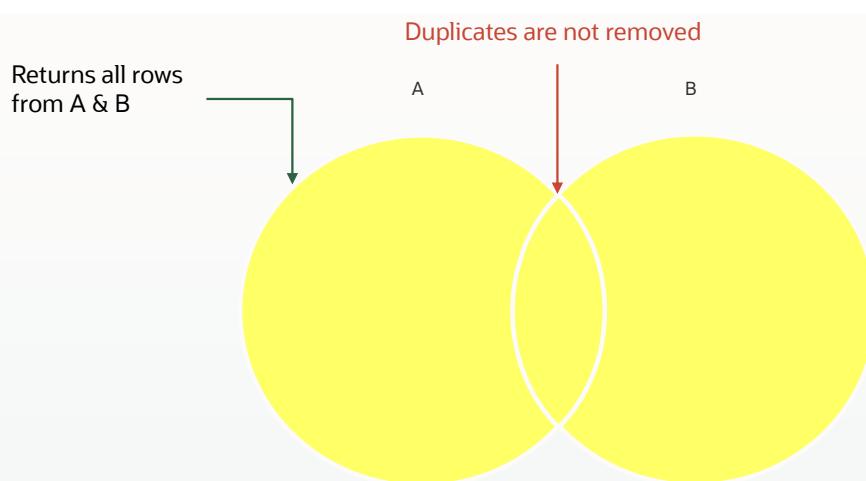
15



O

The UNION operator eliminates any duplicate records. If records that occur in both the EMPLOYEES and the RETIRED\_EMPLOYEES tables are identical, the records are displayed only once.

# UNION ALL Operator



The UNION ALL operator returns rows from both queries, including all duplications.

16

0

Use the UNION ALL operator to return all rows from multiple queries.

## Guidelines

The guidelines for UNION and UNION ALL are the same, with the following two exceptions that pertain to UNION ALL: Unlike UNION, duplicate rows are not eliminated and the output is not sorted by default.

# Using the UNION ALL Operator

Display the jobs and departments of all current and previous employees.

```
SELECT job_id, department_id  
FROM employees  
UNION ALL  
SELECT job_id, department_id  
FROM retired_employees  
ORDER BY job_id;
```

#	JOB_ID	DEPARTMENT_ID
1	AC_ACCOUNT	110
2	AC_MGR	110
3	AD_ASST	10
4	AD_PRES	90
5	AD_PRES	90
6	AD_VP	90
7	AD_VP	80
8	AD_VP	90
9	AD_VP	90



28	SA_REP	80
29	SA_REP	80
30	SA_REP	(null)
31	ST_CLERK	50
32	ST_CLERK	50
33	ST_CLERK	50
34	ST_CLERK	50
35	ST_MAN	50

...

#	job_id	department_id
1	AC_ACCOUNT	110
2	AC_MGR	110
3	AD_ASST	10
4	AD_PRES	90
5	AD_PRES	90
6	AD_VP	90
7	AD_VP	90
8	AD_VP	80
9	AD_VP	90



28	SA_REP	NULL
29	SA_REP	80
30	SA_REP	80
31	ST_CLERK	50
32	ST_CLERK	50
33	ST_CLERK	50
34	ST_CLERK	50
35	ST_MAN	50

0

17

In the example in the slide, 35 rows are selected. The combination of the two tables totals to 35 rows. The UNION ALL operator does not eliminate duplicate rows. UNION returns all distinct rows selected by either query. UNION ALL returns all rows selected by either query, including all duplicates.

Consider the query in the slide, now written with the UNION clause:

```
SELECT job_id,department_id  
FROM employees  
UNION  
SELECT job_id,department_id  
FROM retired_employees  
ORDER BY job_id;
```

The preceding query returns 19 rows. This is because it eliminates all the duplicate rows.

# Lesson Agenda

- Set operators: Types and guidelines
- Tables used in this lesson
- UNION and UNION ALL operator
- INTERSECT operator
- MINUS operator
- Matching SELECT statements
- Using ORDER BY clause in set operations

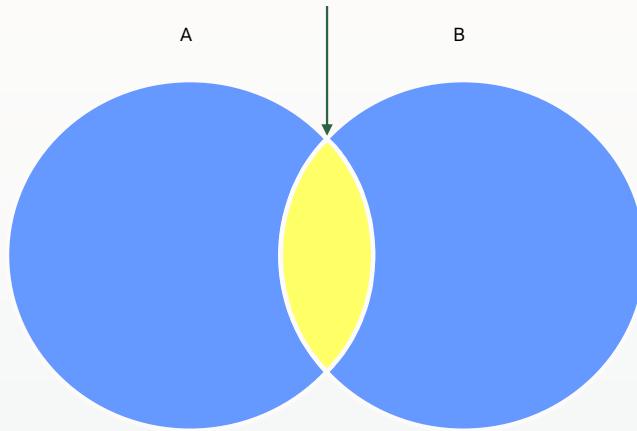


0

For Instructor Use Only.  
This document should not be distributed.

# INTERSECT Operator

Returns all rows common to A & B



The INTERSECT operator returns rows that are common to both queries.

19

0

Use the `INTERSECT` operator to return all rows that are common to multiple queries.

## Guidelines

- The number of columns and the data types of the columns being selected by the `SELECT` statements in the queries must be identical in all the `SELECT` statements used in the query. The names of the columns, however, need not be identical.
- Reversing the order of the intersected tables does not alter the result.
- `INTERSECT` does not ignore `NULL` values.

# Using the INTERSECT Operator

Display the common manager IDs and department IDs of current and previous employees.

```
SELECT manager_id,department_id
FROM employees
INTERSECT
SELECT manager_id,department_id
FROM retired_employees;
```



	MANAGER_ID	DEPARTMENT_ID
1	149	80



20

0

In the example in this slide, the query returns only those records that have the same values in the selected columns in both tables.

# Lesson Agenda

- Set operators: Types and guidelines
- Tables used in this lesson
- UNION and UNION ALL operator
- INTERSECT operator
- MINUS operator
- Matching SELECT statements
- Using the ORDER BY clause in set operations

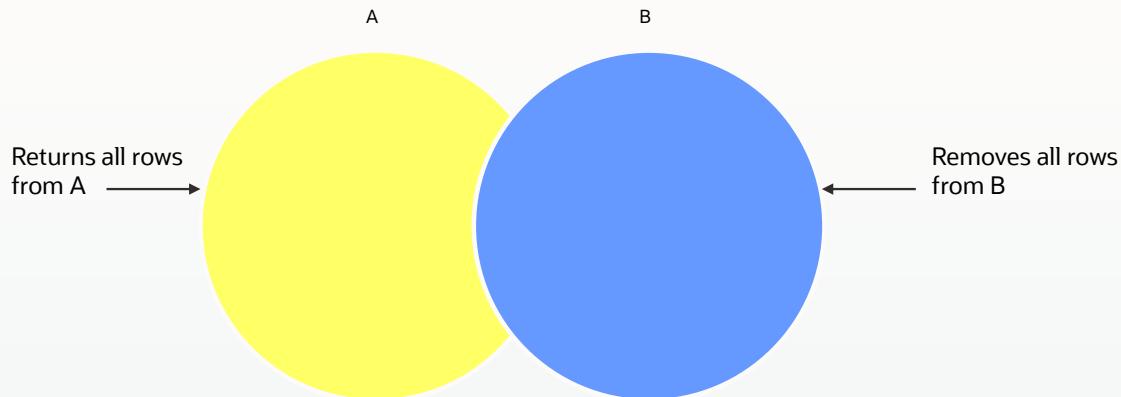


0

For Instructor Use Only.  
This document should not be distributed.



## MINUS Operator



The MINUS operator returns all the distinct rows selected by the first query, but not present in the second query result set.

22

O

Use the MINUS operator to return all distinct rows selected by the first query, but not present in the second query result set (the first SELECT statement MINUS the second SELECT statement).

**Note:** The number of columns must be the same and the data types of the columns being selected by the SELECT statements in the queries must belong to the same data type group in all the SELECT statements used in the query. The names of the columns, however, need not be identical.

# Using the MINUS Operator

Display the manager IDs and Job IDs of employees whose managers have never managed retired employees in the Sales department.

```
SELECT manager_id, job_id  
FROM employees  
WHERE department_id = 80  
MINUS  
SELECT manager_id, job_id  
FROM retired_employees  
WHERE department_id = 80;
```



MANAGER_ID	JOB_ID
1	100 SA_MAN
2	149 SA_REP



O

23

In the example in the slide, the manager IDs and job IDs in the RETIRED\_EMPLOYEES table are subtracted from those in the EMPLOYEES table. The results set displays the employees remaining after the subtraction; they are represented by rows that exist in the EMPLOYEES table, but do not exist in the RETIRED\_EMPLOYEES table. These are the records of the employees who work in the sales department and are being managed by managers who have not managed any of the retired employees.

# Lesson Agenda

- Set operators: Types and guidelines
- Tables used in this lesson
- UNION and UNION ALL operator
- INTERSECT operator
- MINUS operator
- Matching SELECT statements
- Using ORDER BY clause in set operations



0

For Instructor Use Only.  
This document should not be distributed.

## Matching SELECT Statements in Oracle

You must match the data type (using the `TO_CHAR` function or any other conversion functions) when columns do not exist in one or the other table.

```
SELECT location_id, department_name "Department",
       TO_CHAR(NULL) "Warehouse location"
  FROM departments
 UNION
SELECT location_id, TO_CHAR(NULL) "Department",
       state_province
  FROM locations;
```

25

0

Because the expressions in the `SELECT` lists of the queries must match in number, you can use the dummy columns and the data type conversion functions to comply with this rule.

To match the column list explicitly, you can insert `NULL` columns at the missing positions so as to match the count and data type of selected columns in each `SELECT` statement.

In the slide, the name `Warehouse location` is given as the dummy column heading. The `TO_CHAR` function is used in the first query to match the `VARCHAR2` data type of the `state_province` column that is retrieved by the second query. Similarly, the `TO_CHAR` function in the second query is used to match the `VARCHAR2` data type of the `department_name` column that is retrieved by the first query.



# Matching the SELECT Statement: Example in Oracle

Using the UNION operator, display the employee name, job ID, and hire date of all employees.

```
SELECT FIRST_NAME, JOB_ID, hire_date "HIRE_DATE"
FROM employees
UNION
SELECT FIRST_NAME, JOB_ID, TO_DATE(NULL) "HIRE_DATE"
FROM retired_employees;
```



	FIRST_NAME	JOB_ID	HIRE_DATE
1	Alex	PU_CLERK	(null)
2	Alexander	IT_PROG	03-JAN-14
3	Alexandera	IT_PROG	(null)
4	Bruce	IT_PROG	21-MAY-15
5	Bruk	IT_PROG	(null)
6	Curtis	ST_CLERK	29-JAN-13
7	Dany	FI_ACCOUNT	(null)
8	DeI	PU_MAN	(null)

26

O



The EMPLOYEES and RETIRED\_EMPLOYEES tables have several columns in common (for example, EMPLOYEE\_ID, JOB\_ID, and DEPARTMENT\_ID). However, what if you want the query to display FIRST\_NAME, JOB\_ID, and HIRE\_DATE using the UNION operator, knowing that HIRE\_DATE exists only in the EMPLOYEES table?

The code example in the slide matches the FIRST\_NAME and JOB\_ID columns in the EMPLOYEES and RETIRED\_EMPLOYEES tables. NULL is added to the RETIRED\_EMPLOYEES SELECT statement to match the HIRE\_DATE column in the EMPLOYEES SELECT statement.

In the results shown in the slide, each row in the output that corresponds to a record from the RETIRED\_EMPLOYEES table contains a NULL in the HIRE\_DATE column.



# Matching SELECT Statements in MySQL

You must match the data type (using the `CAST` function) when columns do not exist in one or the other table.

```
SELECT location_id, department_name 'Department',
       CAST(NULL AS CHAR) 'Warehouse location'
  FROM departments
 UNION
SELECT location_id, CAST(NULL AS CHAR),
       state_province
  FROM locations;
```

#	location_id	Department	Warehouse location
1	1700	Administration	NULL
2	1800	Marketing	NULL
3	1500	Shipping	NULL
4	1400	IT	NULL
5	2500	Sales	NULL
6	1700	Executive	NULL
7	1700	Accounting	NULL
8	1700	Contracting	NULL
9	1500	NULL	California
10	1800	NULL	Ontario
11	2500	NULL	Oxford
12	1400	NULL	Texas
13	1700	NULL	Washington

27

0

Because the expressions in the `SELECT` lists of the queries must match in number, you can use the dummy columns and the data type conversion functions to comply with this rule.

To match the column list explicitly, you can insert `NULL` columns at the missing positions so as to match the count and data type of selected columns in each `SELECT` statement.

In the slide, the name `Warehouse location` is given as the dummy column heading. The `CAST` function is used in the first query to match the `VARCHAR` data type of the `state_province` column that is retrieved by the second query. Similarly, the `CAST` function in the second query is used to match the `VARCHAR` data type of the `department_name` column that is retrieved by the first query.



# Matching the SELECT Statement: Example in MySQL

Using the `UNION` operator, display the employee name, job ID, and hire date of all employees.

```
SELECT first_name, job_id, hire_date 'Hire Date'  
FROM employees  
UNION  
SELECT first_name, job_id, CAST(NULL AS DATE)  
FROM retired_employees;
```

#	first_name	job_id	Hire Date
1	Steven	AD_PRES	2011-06-17
2	Neena	AD_VP	2009-09-21
3	Lex	AD_VP	2009-01-13
4	Alexander	IT_PROG	2014-01-03
5	Bruce	IT_PROG	2015-05-21
6	Diana	IT_PROG	2015-02-07
7	Kevin	ST_MAN	2015-11-16
8	Trenna	ST_CLERK	2011-10-17
9	Curtis	ST_CLERK	2013-01-29

...

28	Rahul	FI_MGR	NULL
29	Dany	FI_ACCOUNT	NULL
30	James	FI_ACCOUNT	NULL
31	Shana	FI_ACCOUNT	NULL
32	Peter	FI_ACCOUNT	NULL
33	Lui	FI_ACCOUNT	NULL
34	Del	PU_MAN	NULL
35	Alex	PU_CLERK	NULL



28

O

The `employees` and `retired_employees` tables have several columns in common (for example, `employee_id`, `job_id`, and `department_id`). However, what if you want the query to display `first_name`, `job_id`, and `hire_date` using the `UNION` operator, knowing that `hire_date` exists only in the `employees` table?

The code example in the slide matches the `first_name` and `job_id` columns in the `employees` and `retired_employees` tables. `NULL` is added to the `retired_employees` SELECT statement to match the `hire_date` column in the `employees` SELECT statement.

In the results shown in the slide, each row in the output that corresponds to a record from the `retired_employees` table contains a `NULL` in the `hire_date` column.

# Lesson Agenda

- Set operators: Types and guidelines
- Tables used in this lesson
- UNION and UNION ALL operator
- INTERSECT operator
- MINUS operator
- Matching SELECT statements
- Using the ORDER BY clause in set operations



0

For Instructor Use Only.  
This document should not be distributed.



# Using the ORDER BY Clause in Set Operations in Oracle

- The ORDER BY clause can appear only once at the end of the compound query.
- Component queries cannot have individual ORDER BY clauses.
- The ORDER BY clause recognizes only the columns of the first SELECT query.
- By default, the first column of the first SELECT query is used to sort the output in ascending order.

30

O



You can use the ORDER BY clause only once in a compound query. Place the ORDER BY clause at the end of the query. The ORDER BY clause accepts the column name or an alias. By default, the output is sorted in ascending order in the first column of the first SELECT query.

**Note:** The ORDER BY clause does not recognize the column names of the second SELECT query. To avoid confusion over column names, it is a common practice to ORDER BY column positions.

For example, in the following statement, the output will be shown in ascending order of job\_id.

```
SELECT employee_id, job_id, salary  
FROM   employees  
  
UNION  
  
SELECT employee_id, job_id, 0  
FROM   retired_employees  
  
ORDER BY 2;
```

If you omit ORDER BY, by default, the output will be sorted in ascending order of employee\_id. You cannot use the columns from the second query to sort the output.



## Using the ORDER BY Clause in Set Operations in Oracle: Example

Display the employee ID and job ID of all current and retired employees, sorted by job ID.

```
SELECT employee_id, job_id
FROM   employees
UNION
SELECT employee_id, job_id
FROM   retired_employees
ORDER BY 2;
```



	EMPLOYEE_ID	JOB_ID
1		206 AC_ACCOUNT
2		205 AC_MGR
3		200 AD_ASST
4		100 AD_PRES
5		301 AD_PRES
6		101 AD_VP
7		102 AD_VP
8		302 AD_VP
9		303 AD_VP
10		309 FI_ACCOUNT
11		310 FI_ACCOUNT
12		311 FI_ACCOUNT
13		312 FI_ACCOUNT
14		313 FI_ACCOUNT
...		
28		174 SA_REP
29		176 SA_REP
30		178 SA_REP
31		141 ST_CLERK
32		142 ST_CLERK
33		143 ST_CLERK
34		144 ST_CLERK
35		124 ST_MAN

0

For Instructor Use Only.  
This document should not be distributed.

31

In the example shown on the slide, the output is sorted in the ascending order of `job_id`.

**Note:** The `ORDER BY` clause does not recognize the column names of the second `SELECT` query. To avoid confusion over column names, it is a common practice to `ORDER BY` column positions.

If you omit `ORDER BY`, by default, the output will be sorted in ascending order of `employee_id`. You cannot use the columns from the second query to sort the output.



# Using the ORDER BY Clause with UNION in MySQL

- To use an ORDER BY clause to sort the entire UNION result, place the ORDER BY clause only once at the end of the compound query.
- The ORDER BY clause uses the columns of the first SELECT query.
- If a column to be sorted is aliased, the ORDER BY clause must use the alias rather than the column name.

32

O

You can use the ORDER BY clause to sort the entire result set in a compound query. Place the ORDER BY clause at the end of the query. In the ORDER BY clause, you can refer to a column alias or column name from the first SELECT statement. If the sort column is aliased, you must use the alias rather than the column name. You can use the column position in the ORDER BY clause, but this use is deprecated.

**Note:** You can use an ORDER BY clause in each SELECT statement, but it has no effect on the sort order of the result set, but rather is used by the LIMIT clause. To apply an ORDER BY clause to an individual SELECT statement, enclose the statement in parentheses, with the ORDER BY and LIMIT clauses inside the parentheses. For example, the following query selects and combines the first 10 employees from each table and sorts the results by Job ID.

```
(SELECT employee_id, job_id  
FROM   employees ORDER BY employee_id LIMIT 10)  
  
UNION  
  
(SELECT employee_id, job_id  
FROM   retired_employees ORDER BY employee_id LIMIT 10)  
  
ORDER BY job_id;
```



## Using the ORDER BY Clause with UNION: Example in MySQL

Display the employee ID and job ID of all current and retired employees, sorted by job ID.

```
SELECT employee_id, job_id
FROM   employees
UNION
SELECT employee_id, job_id
FROM   retired_employees
ORDER BY job_id;
```



#	employee_id	job_id
1	206	AC_ACCOUNT
2	205	AC_MGR
3	200	AD_ASST
4	100	AD_PRES
5	301	AD_PRES
6	101	AD_VP
7	102	AD_VP
8	303	AD_VP
9	302	AD_VP
...		

28	178	SA_REP
29	176	SA_REP
30	174	SA_REP
31	144	ST_CLERK
32	143	ST_CLERK
33	142	ST_CLERK
34	141	ST_CLERK
35	124	ST_MAN

33

O

In the example shown on the slide, the output is sorted in the ascending order of `job_id`.

For Instructor Use Only.  
This document should not be distributed.

# Summary

In this lesson, you should have learned how to use:

- UNION to return all distinct rows
- UNION ALL to return all rows, including duplicates
- INTERSECT to return all rows that are shared by both queries
- MINUS to return all distinct rows that are selected by the first query, but not by the second
- ORDER BY only at the very end of the statement



- The UNION operator returns all the distinct rows selected by each query in the compound query. Use the UNION operator to return all rows from multiple tables and eliminate any duplicate rows.
- Use the UNION ALL operator to return all rows from multiple queries. Unlike the case with the UNION operator, duplicate rows are not eliminated and the output is not sorted by default.
- Use the INTERSECT operator to return all rows that are common to multiple queries.
- Use the MINUS operator to return rows returned by the first query that are not present in the second query.
- Remember to use the ORDER BY clause only at the very end of the compound statement.
- Make sure that the corresponding expressions in the SELECT lists match in number and data type.

## Practice 9: Overview

In this practice, you create reports by using:

- The UNION operator
- The INTERSECT operator
- The MINUS operator



0

35

In this practice, you write queries using set operators.

For Instructor Use Only.  
This document should not be distributed.

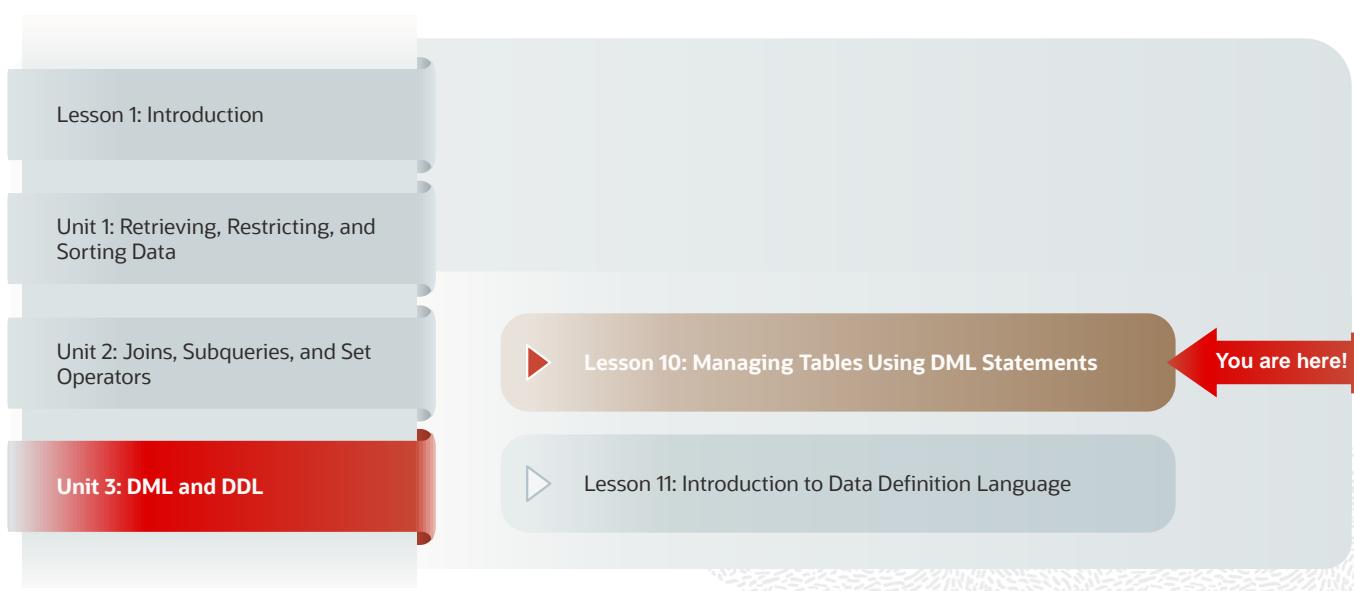
For Instructor Use Only.  
This document should not be distributed.

# Managing Tables Using DML Statements in Oracle

0

For Instructor Use Only.  
This document should not be distributed.

# Course Roadmap



2

In Unit 3, you learn how to manage data in tables using data manipulation language (DML) statements. You also learn how to create and manage database objects using data definition language (DDL) statements.

# Objectives

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

- Describe each data manipulation language (DML) statement
- Control transactions

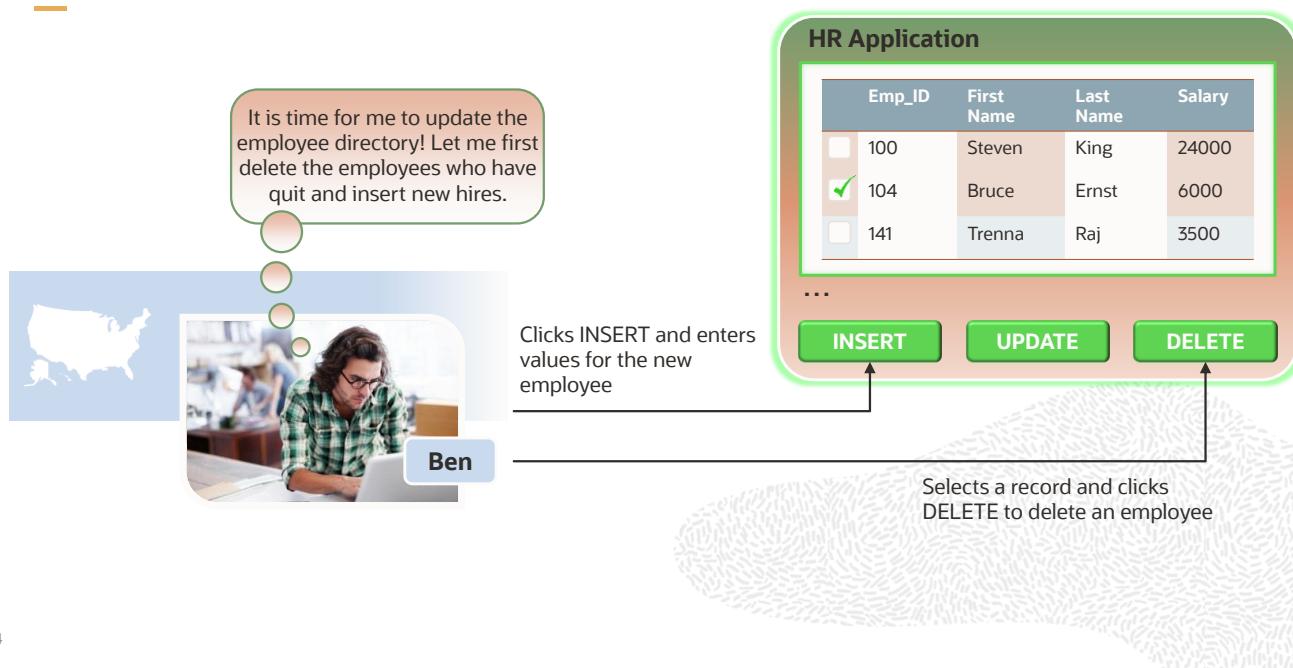


0

3

In this lesson, you learn how to use data manipulation language (DML) statements to insert rows into a table, update existing rows in a table, and delete existing rows from a table. You also learn how to control transactions with the COMMIT, SAVEPOINT, and ROLLBACK statements.

# HR Application Scenario



4

Ben is an HR manager in USA. Ben wants to update the outdated employee list in his organization because he has hired new employees recently and a few employees have left the organization.

Ben logs in to the HR application and selects the ex-employee records and clicks on **DELETE**. He then clicks **INSERT** and enters the details of new hires and clicks **SAVE**. The employee list is now updated.

When the HR manager performs these transactions in the HR application, data manipulation language (DML) statements are used in the background. DML statements modify the data in the tables. In this lesson, you learn about DML statements and how to use them.

# Lesson Agenda

- Adding new rows in a table
  - INSERT statement
- Changing data in a table
  - UPDATE statement
- Removing rows from a table:
  - DELETE statement
  - TRUNCATE statement
- Database transaction control using COMMIT, ROLLBACK, and SAVEPOINT
- Read consistency
- Manual Data Locking
  - FOR UPDATE clause in a SELECT statement
  - LOCK TABLE statement

5

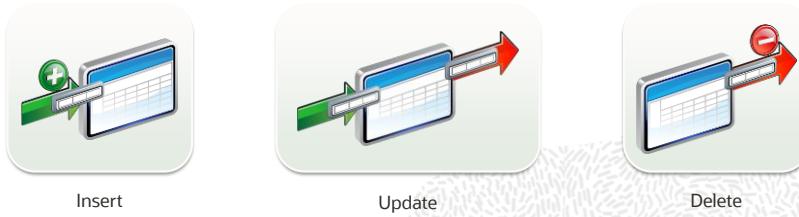
0



For Instructor Use Only.  
This document should not be distributed.

# Data Manipulation Language

- A DML statement is executed when you:
  - Add new rows to a table
  - Modify existing rows in a table
  - Remove existing rows from a table
- A *transaction* consists of a collection of DML statements that form a logical unit of work.



6

O

Data manipulation language (DML) is a core part of SQL. 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*.

Consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction might consist of three separate operations: decreasing the savings account, increasing the checking account, and recording the transaction in the transaction journal. The database server must guarantee that all the three SQL statements are performed to maintain the accounts in proper balance. When something prevents one of the statements in the transaction from executing, the other statements of the transaction must be undone.

**Note:** Most of the DML statements in this lesson assume that no constraints on the table are violated. Constraints are discussed later in this course.

# Adding a New Row to a Table

DEPARTMENTS				70 Public Relations	100	1700	New row
#	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID			
1	10	Administration	200	1700			
2	20	Marketing	201	1800			
3	50	Shipping	124	1500			
4	60	IT	103	1400			
5	80	Sales	149	2500			
6	90	Executive	100	1700			
7	110	Accounting	205	1700			
8	190	Contracting	(null)	1700			

Insert a new row into the DEPARTMENTS table.

#	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	70	Public Relations	100	1700
2	10	Administration	200	1700
3	20	Marketing	201	1800
4	50	Shipping	124	1500
5	60	IT	103	1400
6	80	Sales	149	2500
7	90	Executive	100	1700
8	110	Accounting	205	1700
9	190	Contracting	(null)	1700

7

0

The graphic in the slide illustrates the addition of a new department record to the DEPARTMENTS table.

# INSERT Statement Syntax

- Add new rows to a table by using the `INSERT` statement.

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

- With this syntax, only one row is inserted at a time.



0

8

For Instructor Use Only.  
This document should not be distributed.

You can add new rows to a table by issuing the `INSERT` statement.

In the syntax:

<code>table</code>	Is the name of the table
<code>column</code>	Is the name of the column in the table to populate
<code>value</code>	Is the corresponding value for the column

**Note:** This statement with the `VALUES` clause adds only one row at a time.

# Inserting New Rows

- Insert a new row containing values for each column.
- List values in the default order of the columns in the table.
- Optionally, list the columns in the `INSERT` clause.

```
INSERT INTO departments(department_id,  
                      department_name, manager_id, location_id)  
VALUES (70, 'Public Relations', 100, 1700);
```

```
1 row inserted.
```

- Enclose character and date values within single quotation marks.

9

0

Because you can insert a new row that contains values for each column, the column list is not required in the `INSERT` clause. However, if you do not use the column list, the values must be listed according to the default order of the columns in the table, and a value must be provided for each column.

```
DESCRIBE departments
```

For clarity, use the column list in the `INSERT` clause.

Enclose character and date values within single quotation marks; however, it is not recommended that you enclose numeric values within single quotation marks.

# Inserting Rows with Null Values

- Implicit method: Omit the column from the column list.

```
INSERT INTO departments (department_id,  
                      department_name)  
VALUES      (30, 'Purchasing');
```

1 row inserted.

- Explicit method: Specify the NULL keyword in the VALUES list.

```
INSERT INTO departments  
VALUES      (100, 'Finance', NULL, NULL);
```

1 row inserted.

Method	Description
Implicit	Omit the column from the column list.
Explicit	Specify the NULL keyword in the VALUES list; specify the empty string (' ') in the VALUES list for character strings and dates.

Be sure that you can use null values in the targeted column by verifying the NULL status with the DESCRIBE command.

The Oracle server automatically enforces all data types, data ranges, and data integrity constraints. If a column is not explicitly listed, a null value is inserted in the new row unless you have default values for the missing columns that are used.

Common errors that can occur when you are inserting are in the following order:

- Mandatory value missing for a NOT NULL column
- Duplicate value violating any unique or primary key constraint
- Any value violating a CHECK constraint
- Referential integrity maintained for foreign key constraint
- Data type mismatches or values too wide to fit in a column

**Note:** Use of the column list is recommended because it makes the INSERT statement more readable and reliable, or less prone to mistakes.

# Inserting Special Values

The CURRENT\_DATE function records the current date and time in Oracle.

```
INSERT INTO employees (employee_id,
                      first_name, last_name,
                      email, phone_number,
                      hire_date, job_id, salary,
                      commission_pct, manager_id,
                      department_id)
VALUES (113,
        'Louis', 'Popp',
        'LPOPP', '515.124.4567',
        CURRENT_DATE, 'AC_ACCOUNT', 6900,
        NULL, 205, 110);
```

1 row inserted.

11

You can use functions to enter special values in your table.

The example in the slide records information for employee Popp in the EMPLOYEES table. It supplies the current date and time in the HIRE\_DATE column. It uses the CURRENT\_DATE function that returns the current date in the session time zone. You can also use the USER function when inserting rows in a table. The USER function records the current username.

## Confirming Additions to the Table

```
SELECT employee_id, last_name, job_id, hire_date, commission_pct
FROM   employees
WHERE  employee_id = 113;
```

# Inserting Specific Date and Time Values

- Add a new employee.

```
INSERT INTO employees
VALUES
(114,
'Den', 'Raphealy',
'DRAPHEAL', '515_127_4561',
TO DATE('FEB 3, 2016', 'MON DD, YYYY'),
'SA REP', 11000, 0.2, 100, 70);
```

1 row inserted.

- Verify your addition.

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	114	Den	Raphealy	DRAPEAL	515.127.4561	03-FEB-16	SA REP	11000	0.2	100	70

12

0

The DD-MON-RR format is generally used to insert a date value. With the RR format, the system provides the correct century automatically.

You may also supply the date value in the DD-MON-YYYY format. This is recommended because it clearly specifies the century and does not depend on the internal RR format logic of specifying the correct century.

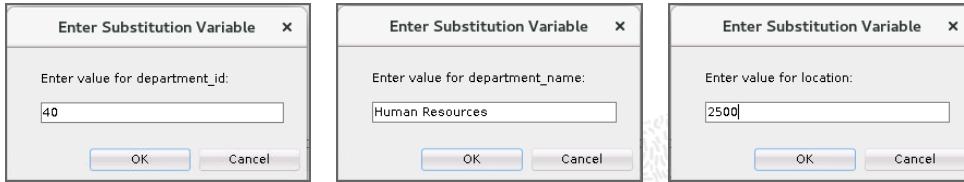
If you want to enter the date in a format other than the default format (for example, with another century or a specific time), you must use the TO\_DATE function.

The example in the slide records information for employee Raphealy in the EMPLOYEES table. It sets the HIRE\_DATE column to be February 3, 2016.

# Creating a Script

- Use the & substitution in a SQL statement to prompt for values.
- & is a placeholder for the variable value.

```
INSERT INTO departments
  (department_id, department_name, location_id)
VALUES (&department_id, '&department_name', &location);
```



13

O

You can save commands with substitution variables to a file and execute the commands in the file. The example in the slide records information for a department in the DEPARTMENTS table.

Run the script file and you are prompted for input for each of the ampersand (&) substitution variables. After entering a value for the substitution variable, click the OK button. The values that you input are then substituted into the statement. This enables you to run the same script file over and over, but supply a different set of values each time you run it.

# Copying Rows from Another Table

- Write your `INSERT` statement with a subquery:

```
INSERT INTO sales_reps(id, name, salary, commission_pct)
SELECT employee_id, last_name, salary, commission_pct
FROM   employees
WHERE  job_id LIKE '%REP%';
```

5 rows inserted.

- Do not use the `VALUES` clause.
- Match the number of columns in the `INSERT` clause to those in the subquery.
- Inserts all the rows returned by the subquery in the table, `sales_reps`.

You can use the `INSERT` statement to add rows to a table where the values are derived from existing tables. In the example in the slide, for the `INSERT INTO` statement to work, you must have already created the `sales_reps` table using the `CREATE TABLE` statement. `CREATE TABLE` is discussed in the lesson titled “Introduction to Data Definition Language.”

In place of the `VALUES` clause, you use a subquery.

## Syntax

```
INSERT INTO table [ column (, column) ] subquery;
```

In the syntax:

<code>table</code>	Is the name of the table
<code>column</code>	Is the name of the column in the table to populate
<code>subquery</code>	Is the subquery that returns rows to the table

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. Zero or more rows are added depending on the number of rows returned by the subquery. To create a copy of the rows of a table, use `SELECT *` in the subquery:

```
INSERT INTO copy_emp
SELECT *
FROM   employees;
```

# Lesson Agenda

- Adding new rows in a table
  - `INSERT` statement
- Changing data in a table
  - `UPDATE` statement
- Removing rows from a table:
  - `DELETE` statement
  - `TRUNCATE` statement
- Database transaction control using `COMMIT`, `ROLLBACK`, and `SAVEPOINT`
- Read consistency
- Manual Data Locking
  - `FOR UPDATE` clause in a `SELECT` statement
  - `LOCK TABLE` statement



0

For Instructor Use Only.  
This document should not be distributed.

# Changing Data in a Table

EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	MANAGER_ID	COMMISSION_PCT	DEPARTMENT_ID
100	Steven	King	24000	(null)	(null)	90
101	Neena	Kochhar	17000	100	(null)	90
102	Lex	De Haan	17000	100	(null)	90
103	Alexander	Hunold	9000	102	(null)	60
104	Bruce	Ernst	6000	103	(null)	60
107	Diana	Lorentz	4200	103	(null)	60
124	Kevin	Mourgos	5800	100	(null)	50

Update rows in the EMPLOYEES table:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	MANAGER_ID	COMMISSION_PCT	DEPARTMENT_ID
100	Steven	King	24000	(null)	(null)	90
101	Neena	Kochhar	17000	100	(null)	90
102	Lex	De Haan	17000	100	(null)	90
103	Alexander	Hunold	9000	102	(null)	80
104	Bruce	Ernst	6000	103	(null)	80
107	Diana	Lorentz	4200	103	(null)	80
124	Kevin	Mourgos	5800	100	(null)	50

16

0

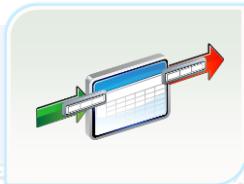
The slide illustrates changing the department number for employees in department 60 to department 80.

# UPDATE Statement Syntax

- Modify existing values in a table with the UPDATE statement:

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

- Update more than one row at a time (if required).



0

17

You can modify the existing values in a table by using the UPDATE statement.

In the syntax:

<i>table</i>	Is the name of the table
<i>column</i>	Is the name of the column in the table to populate
<i>value</i>	Is the corresponding value or subquery for the column
<i>Condition</i> names, comparison operators	Identifies the rows to be updated and is composed of column expressions, constants, subqueries, and

Confirm the update operation by querying the table to display the updated rows.

For more information, see the section on “UPDATE” in *Oracle Database SQL Language Reference* for 19c database.

**Note:** In general, use the primary key column in the WHERE clause to identify a single row for update. Using other columns can unexpectedly cause several rows to be updated. For example, identifying a single row in the EMPLOYEES table by last name may return more than one employee having the same last name.

# Updating Rows in a Table

- Values for a specific row or rows are modified if you specify the WHERE clause:

```
UPDATE employees
SET department_id = 50
WHERE employee_id = 113;
```

1 row updated.

- Values for all the rows in the table are modified if you omit the WHERE clause:

```
UPDATE copy_emp
SET department_id = 110;
22 rows updated
```

- Specify SET column\_name= NULL to update a column value to NULL.

The UPDATE statement modifies the values of a specific row or rows if the WHERE clause is specified. The example in the slide shows the transfer of employee 113 (Popp) to department 50.

If you omit the WHERE clause, values for all the rows in the table are modified. Examine the updated rows in the COPY\_EMP table.

```
SELECT last_name, department_id
FROM copy_emp;
```

For example, an employee who was an SA REP has now changed his job to an IT PROG. Therefore, his JOB\_ID needs to be updated and the commission field needs to be set to NULL.

```
UPDATE employees
SET job_id = 'IT_PROG', commission_pct = NULL
WHERE employee_id = 114;
```

**Note:** The COPY\_EMP table has the same data as the EMPLOYEES table.

# Updating Two Columns with a Subquery

Update employee 103's job and salary to match those of employee 205.

```
UPDATE employees
SET (job_id,salary) = (SELECT job_id,salary
                         FROM employees
                           WHERE employee_id = 205)
      WHERE employee_id = 103;
```

1 row updated.

19

You can update multiple columns in the SET clause of an UPDATE statement by writing multiple subqueries. The syntax is as follows:

```
UPDATE table
SET column =
        (SELECT column
         FROM table
           WHERE condition)
      [ ,
        column =
        (SELECT column
         FROM table
           WHERE condition) ]
[WHERE condition ] ;
```

# Updating Rows Based on Another Table

Use the subqueries in the UPDATE statements to update row values in a table based on values from another table:

```
UPDATE copy_emp
SET department_id = (SELECT department_id
                      FROM employees
                      WHERE employee_id = 100)
WHERE job_id      = (SELECT job_id
                      FROM employees
                      WHERE employee_id = 200);
1 row updated.
```

20

0

You can use the subqueries in the UPDATE statements to update values in a table. The example in the slide updates the COPY\_EMP table based on the values from the EMPLOYEES table. It changes the department number to employee 100's current department number for all the employees whose job ID is the same as employee 200's job ID.

# Lesson Agenda

- Adding new rows in a table
  - INSERT statement
- Changing data in a table
  - UPDATE statement
- Removing rows from a table:
  - DELETE statement
  - TRUNCATE statement
- Database transaction control using COMMIT, ROLLBACK, and SAVEPOINT
- Read consistency
- Manual Data Locking
  - FOR UPDATE clause in a SELECT statement
  - LOCK TABLE statement

21

0



For Instructor Use Only.  
This document should not be distributed.

# Removing a Row from a Table

DEPARTMENTS

#	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700
8	190	Contracting	(null)	1700

Delete a row from the DEPARTMENTS table:

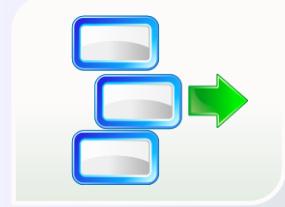
#	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700

The contracting department has been removed from the DEPARTMENTS table (assuming no constraints on the DEPARTMENTS table are violated), as shown in the graphic in the slide.

# DELETE Statement

You can remove existing rows from a table by using the `DELETE` statement:

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



23

0

In the syntax:

*table* Is the name of the table

*condition* Identifies the rows to be deleted, and is composed of column names, expressions, constants, subqueries, and comparison operators

**Note:** If no rows are deleted, the message “0 rows deleted” is returned (on the Script Output tab in SQL Developer).

For more information, see the section on “`DELETE`” in *Oracle Database SQL Language Reference* for 19c database.

# Deleting Rows from a Table

- Specific rows are deleted if you specify the WHERE clause:

```
DELETE FROM departments  
WHERE department_name = 'Finance';  
1 row deleted.
```

- All rows in the table are deleted if you omit the WHERE clause:

```
DELETE FROM copy_emp;  
22 rows deleted
```

24

0

You can delete specific rows by specifying the WHERE clause in the DELETE statement. The first example in the slide deletes the accounting department from the DEPARTMENTS table. You can confirm the delete operation by trying to display the deleted rows using the SELECT statement. The query returns 0 rows.

```
SELECT *  
FROM departments  
WHERE department_name = 'Finance';
```

However, if you omit the WHERE clause, all rows in the table are deleted. The second example in the slide deletes all rows from the COPY\_EMP table, because no WHERE clause was specified.

## Example

Remove rows identified in the WHERE clause.

```
DELETE FROM employees WHERE employee_id = 113;
```

```
DELETE FROM departments WHERE department_id IN (30, 40);
```

# Deleting Rows Based on Another Table

Use the subqueries in the `DELETE` statements to remove rows from a table based on values from another table:

```
DELETE FROM employees
WHERE department_id IN
    (SELECT department_id
     FROM departments
     WHERE department_name
          LIKE '%Public%');

1 row deleted.
```

You can use the subqueries to delete rows from a table based on values from another table. The example in the slide deletes all the employees in a department, where the department name contains the string `Public`.

The subquery searches the `DEPARTMENTS` table to find the department number based on the department name containing the string `Public`. The subquery then feeds the department number to the main query, which deletes rows of data from the `EMPLOYEES` table.

# TRUNCATE Statement

- Removes all rows from a table, leaving the table empty and the table structure intact
- Is a data definition language (DDL) statement rather than a DML statement; cannot be undone
- Syntax:

```
TRUNCATE TABLE table_name;
```

- Example:

```
TRUNCATE TABLE copy_emp;
```

You can use the TRUNCATE statement to quickly remove all rows from a table or cluster efficiently. Removing rows with the TRUNCATE statement is faster than removing them with the DELETE statement for the following reasons:

- The TRUNCATE statement is a data definition language (DDL) statement and generates no rollback information. Rollback information is covered later in this lesson.

If the table is the parent of a referential integrity constraint, you cannot truncate the table. You need to disable the constraint before issuing the TRUNCATE statement. You will learn more about DDL statements and disabling constraints in the lesson titled “*Introduction to Data Definition Language*.”

# Lesson Agenda

- Adding new rows in a table
  - INSERT statement
- Changing data in a table
  - UPDATE statement
- Removing rows from a table:
  - DELETE statement
  - TRUNCATE statement
- Database transaction control using COMMIT, ROLLBACK, and SAVEPOINT
- Read consistency
- Manual Data Locking
  - FOR UPDATE clause in a SELECT statement
  - LOCK TABLE statement

27

0



For Instructor Use Only.  
This document should not be distributed.

# Database Transactions

A database transaction consists of one of the following:

- DML statements that constitute one consistent change to the data
- One DDL statement
- One data control language (DCL) statement



0

28

Transactions give you more flexibility and control when changing data, and the Oracle server ensures data consistency in the event of user process failure or system failure.

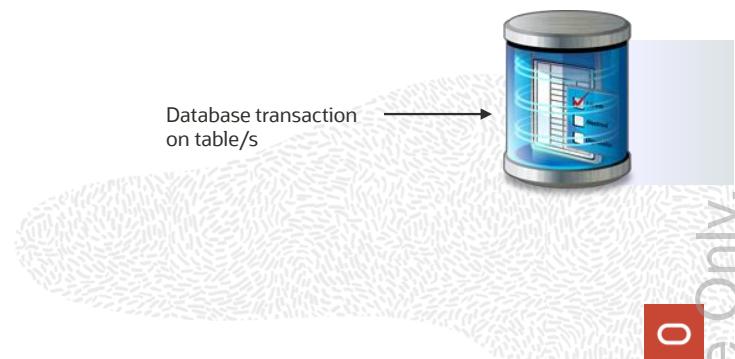
Transactions consist of DML statements that constitute one consistent change to the data. For example, a transfer of funds between two accounts should include the debit in one account and the credit to another account of the same amount. Both actions should either fail or succeed together; the credit should not be committed without the debit.

## Transaction Types

Type	Description
Data manipulation language (DML)	Consists of any number of DML statements that the Oracle server treats as a single entity or a logical unit of work
Data definition language (DDL)	Consists of only one DDL statement
Data control language (DCL)	Consists of only one DCL statement

# Database Transactions: Start and End

- Begin when the first DML SQL statement is executed
- End with one of the following events:
  - A COMMIT or ROLLBACK statement is issued.
  - A DDL or DCL statement executes (automatic commit).
  - The user exits SQL Developer or SQL\*Plus.
  - The system crashes.



29

When does a database transaction start and end?

A transaction begins when the first DML statement is encountered and ends when one of the following occurs:

- A COMMIT or ROLLBACK statement is issued.
- A DDL statement, such as CREATE, is issued.
- A DCL statement is issued.
- The user exits SQL Developer or SQL\*Plus.
- A machine fails or the system crashes.

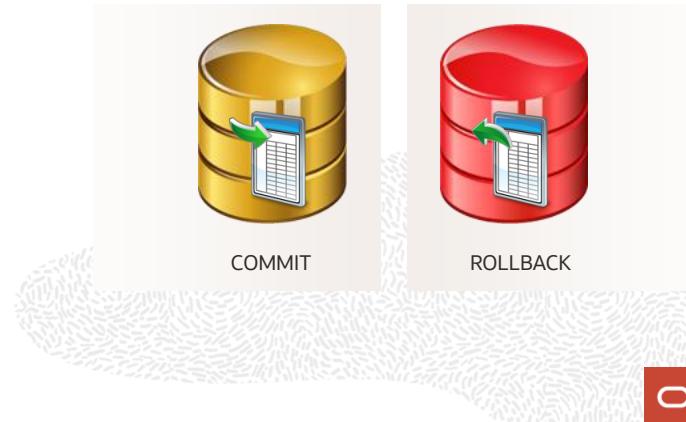
After one transaction ends, the next executable SQL statement automatically starts the next transaction.

A DDL statement or a DCL statement is automatically committed and, therefore, implicitly ends a transaction.

# Advantages of the COMMIT and ROLLBACK Statements

Using COMMIT and ROLLBACK statements, you can:

- Ensure data consistency
- Preview data changes before making changes permanent
- Group logically related operations



30

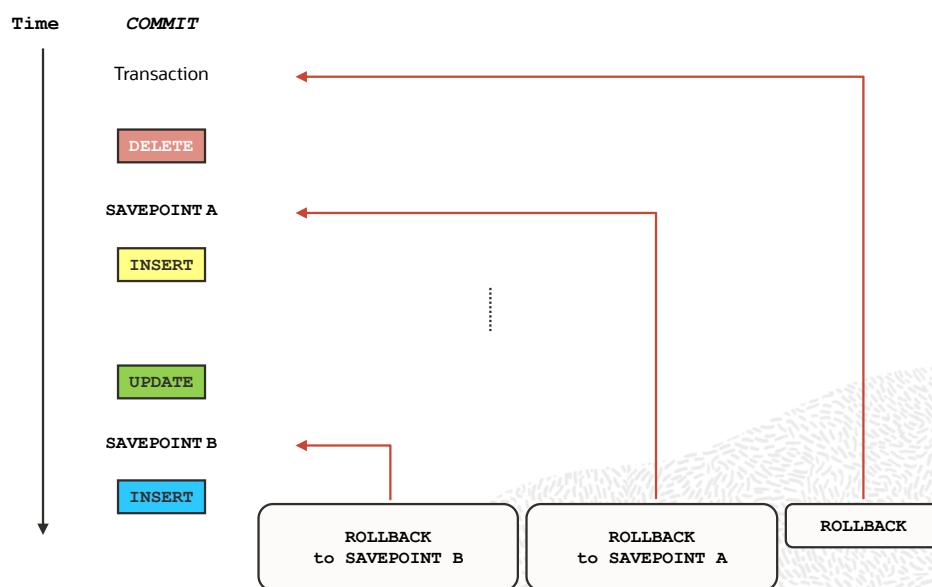
0

With the COMMIT and ROLLBACK statements, you have control over making changes to the data permanent.

# Explicit Transaction Control Statements

31

0



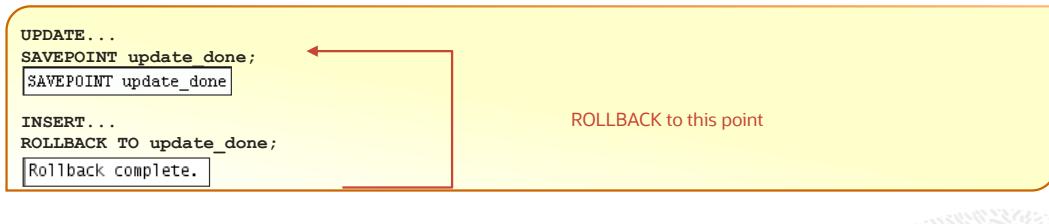
You can control the logic of transactions by using the COMMIT, SAVEPOINT, and ROLLBACK statements.

Statement	Description
COMMIT	COMMIT ends the current transaction by making all pending data changes permanent.
SAVEPOINT <i>name</i>	SAVEPOINT <i>name</i> marks a savepoint within the current transaction.
ROLLBACK	ROLLBACK ends the current transaction by discarding all pending data changes.
ROLLBACK TO SAVEPOINT <i>name</i>	ROLLBACK TO <savepoint> rolls back the current transaction to the specified savepoint, thereby discarding any changes and/or savepoints that were created after the savepoint to which you are rolling back. If you omit the TO SAVEPOINT clause, the ROLLBACK statement rolls back the entire transaction. Because savepoints are logical, there is no way to list the savepoints that you have created.

**Note:** You cannot COMMIT to a SAVEPOINT. SAVEPOINT is not ANSI-standard SQL.

# Rolling Back Changes to a Marker

- Create a marker in the current transaction by using the `SAVEPOINT` statement.
- Roll back to that marker by using the `ROLLBACK TO SAVEPOINT` statement.



32

0

You can create a marker in the current transaction by using the `SAVEPOINT` statement, which divides the transaction into smaller sections. You can then discard pending changes back to that marker by using the `ROLLBACK TO SAVEPOINT` statement.

Note that if you create a second savepoint with the same name as an earlier savepoint, the earlier savepoint is deleted.

# Implicit Transaction Processing

- An automatic commit occurs in the following circumstances:
  - A DDL statement is issued
  - A DCL statement is issued
  - A normal exit from SQL Developer or SQL\*Plus, without explicitly issuing COMMIT or ROLLBACK statements
- An automatic rollback occurs when there is an abnormal termination of SQL Developer or SQL\*Plus, or a system failure.

33

O

Status	Circumstances
Automatic commit	A DDL statement or DCL statement is issued; SQL Developer or SQL*Plus exited normally, without explicitly issuing COMMIT or ROLLBACK commands.
Automatic rollback	Abnormal termination of SQL Developer or SQL*Plus, or system failure.

**Note:** In SQL\*Plus, the AUTOCOMMIT command can be toggled ON or OFF. If set to ON, each individual DML statement is committed as soon as it is executed. You cannot roll back the changes. If set to OFF, the COMMIT statement can still be issued explicitly. Also, the COMMIT statement is issued when a DDL statement is issued or when you exit SQL\*Plus. The SET AUTOCOMMIT ON/OFF command is skipped in SQL Developer. DML is committed on a normal exit from SQL Developer only if you have the Autocommit preference enabled. To enable Autocommit, perform the following:

- From the Tools menu, select Preferences. In the Preferences dialog box, expand Database and select Worksheet Parameters.
- In the right pane, select the “Autocommit in SQL Worksheet” option. Click OK.

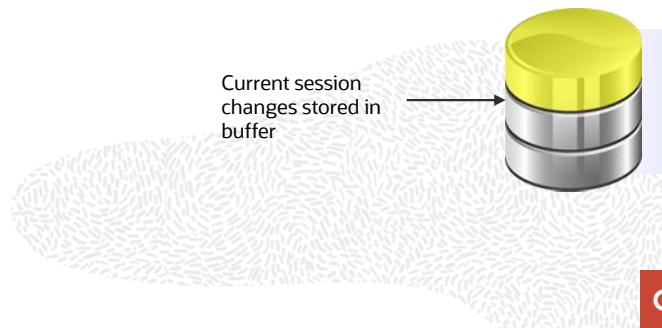
## System Failures

When a transaction is interrupted by a system failure, the entire transaction is automatically rolled back. This prevents the error from causing unwanted changes to the data and returns the tables to the state at the time of the last commit. In this way, the Oracle server protects the integrity of the tables.

In SQL Developer, a normal exit from the session is accomplished by selecting Exit from the File menu. In SQL\*Plus, a normal exit is accomplished by entering the EXIT command at the prompt. Closing the window is interpreted as an abnormal exit.

# State of Data Before COMMIT or ROLLBACK

- You can recover the data of the previous state.
- You can review the results of the DML operations by using the SELECT statement in the current session.
- Other sessions *cannot* view the results of the DML statements issued by the current session.
- The affected rows are locked; other sessions cannot change the data in the affected rows.



34

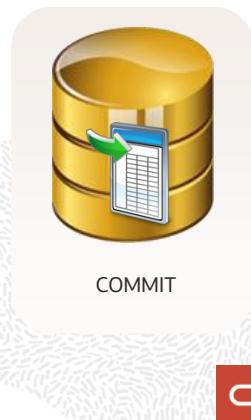
Every data change made during the transaction is temporary until the transaction is committed.

The state of the data before COMMIT or ROLLBACK statements are issued can be described as follows:

- Data manipulation operations primarily affect the database buffer; therefore, the previous state of the data can be recovered.
- The current session can review the results of the data manipulation operations by querying the tables.
- Other sessions cannot view the results of the data manipulation operations made by the current session. The Oracle server institutes read consistency to ensure that each session sees data as it existed at the last commit.
- The affected rows are locked; other sessions cannot change the data in the affected rows.

## State of Data After COMMIT

- Data changes are saved in the database.
- The previous state of the data is overwritten.
- All sessions can view the results.
- Locks on the affected rows are released; those rows are available for other sessions to manipulate.
- All savepoints are erased.



0

35

Make all pending changes permanent by using the `COMMIT` statement. The following happens after a `COMMIT` statement:

- Data changes are written to the database.
- The previous state of the data is no longer available with normal SQL queries.
- All sessions can view the results of the transaction.
- The locks on the affected rows are released; the rows are now available for other sessions to perform new data changes.
- All savepoints are erased.

# Committing Data

- Make the changes:

```
DELETE FROM employees  
WHERE employee_id = 113;  
1 row deleted.  
  
INSERT INTO departments  
VALUES (290, 'Corporate Tax', NULL, 1700);  
1 row inserted.
```

- Commit the changes:

```
COMMIT;  
Commit complete.
```

In the example in the slide, a row is deleted from the EMPLOYEES table and a new row is inserted into the DEPARTMENTS table. The changes are saved by issuing the COMMIT statement.

## Example

Remove departments 290 and 300 from the DEPARTMENTS table and update a row in the EMPLOYEES table. Save the data change.

```
DELETE FROM departments  
WHERE department_id IN (290, 300);  
  
UPDATE employees  
SET department_id = 80  
WHERE employee_id = 206;  
  
COMMIT;
```

## State of Data After ROLLBACK

Discard all pending changes by using the ROLLBACK statement:

- Data changes are undone.
- Previous state of the data is restored.
- Locks on the affected rows are released.

```
DELETE FROM copy_emp;  
ROLLBACK ;
```



ROLLBACK

0

37

Discard all pending changes by using the ROLLBACK statement, which results in the following:

- Data changes are undone.
- The previous state of the data is restored.
- Locks on the affected rows are released.

## State of Data After ROLLBACK: Example

```
DELETE FROM test;
4 rows deleted.

ROLLBACK;
Rollback complete.

DELETE FROM test WHERE id = 100;
1 row deleted.

SELECT * FROM test WHERE id = 100;
No rows selected.

COMMIT;
Commit complete.
```

While attempting to remove a record from the TEST table, you may accidentally empty the table. However, you can correct the mistake by rolling back, reissue a proper statement, and make the data change permanent with COMMIT statement.

**Note:** Refer to code\_ex\_10.sql in the labs/sql1\_oracle/code\_ex folder to create the test table.

# Statement-Level Rollback

- If a single DML statement fails during execution, only that statement is rolled back.
- The Oracle server implements an implicit savepoint.
- All other changes are retained.
- The user should terminate transactions explicitly by executing a COMMIT or ROLLBACK statement.



0

39

If a single DML statement fails during the execution of a transaction, its effect is undone by a statement-level implicit rollback; however, the changes made by the previous DML statements in the transaction are not discarded. These can be committed or rolled back explicitly by the user.

The Oracle server issues an implicit commit before and after any DDL statement. So, even if your DDL statement does not execute successfully, you cannot roll back the previous statement because the server issued a commit.

Terminate your transactions explicitly by executing a COMMIT or ROLLBACK statement.

# Lesson Agenda

- Adding new rows in a table
  - INSERT statement
- Changing data in a table
  - UPDATE statement
- Removing rows from a table:
  - DELETE statement
  - TRUNCATE statement
- Database transaction control using COMMIT, ROLLBACK, and SAVEPOINT
- Read consistency
- Manual Data Locking
  - FOR UPDATE clause in a SELECT statement
  - LOCK TABLE statement



0

For Instructor Use Only.  
This document should not be distributed.

# Read Consistency

- Read consistency guarantees a consistent view of data at all times.
- Changes made by one user do not conflict with the changes made by another user.
- Read consistency ensures that, on the same data:
  - Readers do not wait for writers
  - Writers do not wait for readers
  - Writers wait for writers



41

O

Database users access the database in two ways:

- Read operations (`SELECT` statement)
- Write operations (`INSERT`, `UPDATE`, `DELETE` statements)

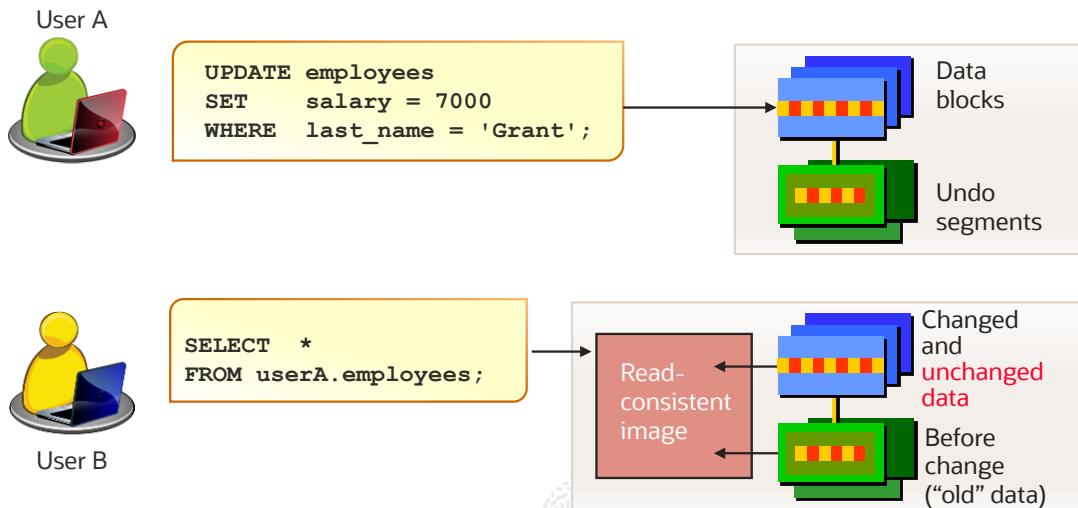
You need read consistency so that:

- The database reader and writer are ensured a consistent view of data
- Readers do not view data that is in the process of being changed
- Writers are ensured that the changes to the database are done in a consistent manner
- Changes made by one writer do not disrupt or conflict with the changes being made by another writer

The purpose of read consistency is to ensure that each user sees data as it existed at the last commit, before a DML operation started.

**Note:** The same user can log in to different sessions. Each session maintains read consistency in the manner described above, even if they are the same users.

# Implementing Read Consistency



42

O

Read consistency is an automatic implementation. It keeps a partial copy of the database in the undo segments. The read-consistent image is constructed from the committed data in the table and the old data that is being changed and is not yet committed from the undo segment.

When an insert, update, or delete operation is made on the database, the Oracle server takes a copy of the data before it is changed and writes it to an *undo segment*.

All readers, except the one who issued the change, see the database as it existed before the changes started; they view the undo segment's "snapshot" of the data.

Before the changes are committed to the database, only the user who is modifying the data sees the database with the alterations. Everyone else sees the snapshot in the undo segment. This guarantees that readers read consistent data that is not currently undergoing change.

When a DML statement is committed, the change made to the database becomes visible to anyone issuing a `SELECT` statement *after* the commit is done. The space occupied by the *old* data in the undo segment file is freed for reuse.

If the transaction is rolled back, the changes are undone:

- The original, older version of the data in the undo segment is written back to the table.
- All users see the database as it existed before the transaction began.

# Lesson Agenda

- Adding new rows in a table
  - INSERT statement
- Changing data in a table
  - UPDATE statement
- Removing rows from a table:
  - DELETE statement
  - TRUNCATE statement
- Database transaction control using COMMIT, ROLLBACK, and SAVEPOINT
- Read consistency
- Manual Data Locking
  - FOR UPDATE clause in a SELECT statement
  - LOCK TABLE statement

43

0



For Instructor Use Only.  
This document should not be distributed.

# FOR UPDATE Clause in a SELECT Statement

- Locks the rows in the EMPLOYEES table where job\_id is SA\_REP.

```
SELECT employee_id, salary, commission_pct, job_id  
FROM employees  
WHERE job_id = 'SA_REP'  
FOR UPDATE  
ORDER BY employee_id;
```

- Lock is released only when you issue a ROLLBACK or a COMMIT.
- If the SELECT statement attempts to lock a row that is locked by another user, the database waits until the row is available, and then returns the results of the SELECT statement.

When you issue a SELECT statement against the database to query some records, no locks are placed on the selected rows. In general, this is required because the number of records locked at any given time is (by default) kept to the absolute minimum: only those records that have been changed but not yet committed are locked. Even then, others will be able to read those records as they appeared before the change (the “before image” of the data). There are times, however, when you may want to lock a set of records even before you change them in your program. Oracle offers the FOR UPDATE clause of the SELECT statement to perform this locking.

When you issue a SELECT...FOR UPDATE statement, RDBMS automatically obtains exclusive row-level locks on all the rows identified by the SELECT statement, thereby holding the records “for your changes only.” No one else will be able to change any of these records until you perform a ROLLBACK or a COMMIT.

You can append the optional keyword NOWAIT to the FOR UPDATE clause to tell the Oracle server not to wait if the table has been locked by another user. In this case, control will be returned immediately to your program or to your SQL Developer environment so that you can perform other work, or simply wait for a period of time before trying again. Without the NOWAIT clause, your process will block until the table is available, when the locks are released by the other user through the issue of a COMMIT or a ROLLBACK command.

## FOR UPDATE Clause: Examples

- You can use the FOR UPDATE clause in a SELECT statement against multiple tables.

```
SELECT e.employee_id, e.salary, e.commission_pct
FROM employees e JOIN departments d
USING (department_id)
WHERE job_id = 'ST_CLERK'
AND location_id = 1500
FOR UPDATE
ORDER BY e.employee_id;
```

- Rows from both the EMPLOYEES and DEPARTMENTS tables are locked.
- Use FOR UPDATE OF *column\_name* to qualify the column that you intend to change; then only the rows from that specific table are locked.

In the example in the slide, the statement locks rows in the EMPLOYEES table with JOB\_ID set to ST\_CLERK and LOCATION\_ID set to 1500, and locks rows in the DEPARTMENTS table with departments in LOCATION\_ID set as 1500.

You can use the FOR UPDATE OF *column\_name* to qualify the column that you intend to change. The OF list of the FOR UPDATE clause does not restrict you to changing only those columns of the selected rows. Locks are still placed on all rows; if you simply state FOR UPDATE in the query and do not include one or more columns after the OF keyword, the database will lock all identified rows across all the tables listed in the FROM clause.

The following statement locks only those rows in the EMPLOYEES table with ST\_CLERK located in LOCATION\_ID 1500. No rows are locked in the DEPARTMENTS table:

```
SELECT e.employee_id, e.salary, e.commission_pct
FROM employees e JOIN departments d
USING (department_id)
WHERE job_id = 'ST_CLERK' AND location_id = 1500
FOR UPDATE OF e.salary
ORDER BY e.employee_id;
```

In the following example, the database is instructed to wait for five seconds for the row to become available, and then return control to you.

```
SELECT employee_id, salary, commission_pct, job_id
FROM employees
WHERE job_id = 'SA_REP'
FOR UPDATE WAIT 5
ORDER BY employee_id;
```

# LOCK TABLE Statement

- Use the `LOCK TABLE` statement to lock one or more tables in a specified mode.
- This manually overrides automatic locking.
- Tables are locked until you `COMMIT` or `ROLLBACK`.

```
LOCK TABLE table_name  
IN [ROW SHARE/ROW EXCLUSIVE/SHARE UPDATE/SHARE/  
     SHARE ROW EXCLUSIVE/ EXCLUSIVE] MODE  
[NOWAIT];
```



O

46

You can use the `LOCK TABLE` statement to manually lock the tables and override automatic locking. The table must be in your schema or you must have the `LOCK ANY TABLE` privilege.

The `lockmode` clause:

SHARE	Permits concurrent queries but prevents update on the locked table
EXCLUSIVE	Permits queries on the locked table but prevents any other activity

For more information about the other `lockmode` clauses, refer to the `LOCK TABLE` statement in *Oracle Database SQL Language Reference* for 19c database.

You can append the optional keyword `NOWAIT` to the `LOCK TABLE` statement to tell the Oracle server not to wait if the table has been locked by another user.

For example, the following statement locks the `EMPLOYEES` table in exclusive mode but does not wait if another user has already locked the table.

```
LOCK TABLE employees  
IN EXCLUSIVE MODE  
NOWAIT;
```

# Summary

In this lesson, you should have learned how to use the following statements:

Function	Description
INSERT	Adds a new row to the table
UPDATE	Modifies existing rows in the table
DELETE	Removes existing rows from the table
TRUNCATE	Removes all rows from a table
COMMIT	Makes all pending changes permanent
SAVEPOINT	Is used to roll back to the savepoint marker
ROLLBACK	Discards all pending data changes
FOR UPDATE clause in SELECT	Locks rows identified by the SELECT query

47

In this lesson, you should have learned how to manipulate data in the Oracle database by using the INSERT, UPDATE, DELETE, and TRUNCATE statements, as well as how to control data changes by using the COMMIT, SAVEPOINT, and ROLLBACK statements. You should have also learned how to use the FOR UPDATE clause of the SELECT statement to lock rows for your changes only.

Remember that the Oracle server guarantees a consistent view of data at all times.

# Practice 10a: Overview

This practice covers the following topics:

- Inserting rows into the tables
- Updating and deleting rows in the table
- Controlling transactions



0

48

In this practice, you add rows to the `MY_EMPLOYEE` table, update and delete data from the table, and control your transactions. You run a script to create the `MY_EMPLOYEE` table.

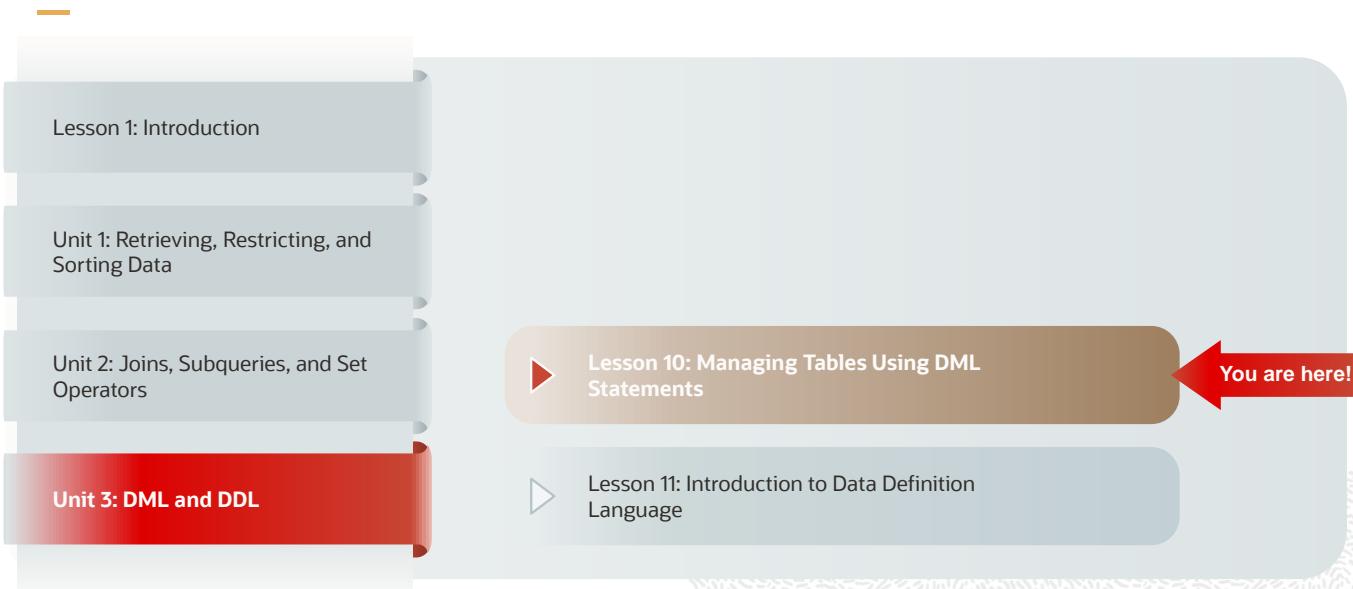
For Instructor Use Only.  
This document should not be distributed.

# Managing Tables Using DML Statements in MySQL

0

For Instructor Use Only.  
This document should not be distributed.

# Course Roadmap



2

0

In Unit 3, you learn how to manage data in tables using data manipulation language (DML) statements. You also learn how to create and manage database objects using data definition language (DDL) statements.

# Objectives

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

- Describe each data manipulation language (DML) statement
- Control transactions

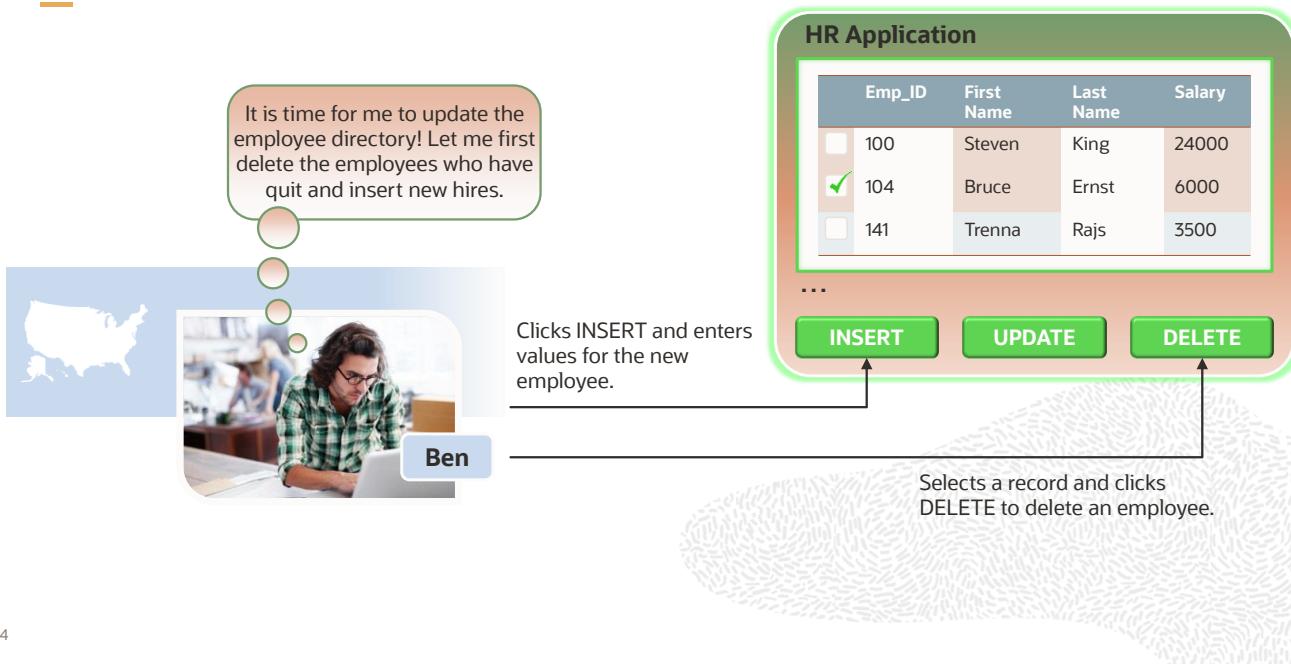


0

3

In this lesson, you learn how to use data manipulation language (DML) statements to insert rows into a table, update existing rows in a table, and delete existing rows from a table. You also learn how to control transactions with the COMMIT, SAVEPOINT, and ROLLBACK statements.

# HR Application Scenario



4

Ben is an HR manager in USA. Ben wants to update the outdated employee list in his organization because he has hired new employees recently and a few employees have left the organization.

Ben logs in to the HR application and selects the ex-employee records and clicks on **DELETE**. He then clicks **INSERT** and enters the details of new hires and clicks **SAVE**. The employee list is now updated.

When the HR manager performs these transactions in the HR application, data manipulation language (DML) statements are used in the background. DML statements modify the data in the tables. In this lesson, you learn about DML statements and how to use them.

# Lesson Agenda

- Adding new rows in a table
  - INSERT statement
- Changing data in a table
  - UPDATE statement
- Removing rows from a table:
  - DELETE statement
  - TRUNCATE statement
- Database transaction control using COMMIT, ROLLBACK, and SAVEPOINT
- Read consistency
- Manual Data Locking
  - FOR UPDATE clause in a SELECT statement
  - LOCK TABLE statement

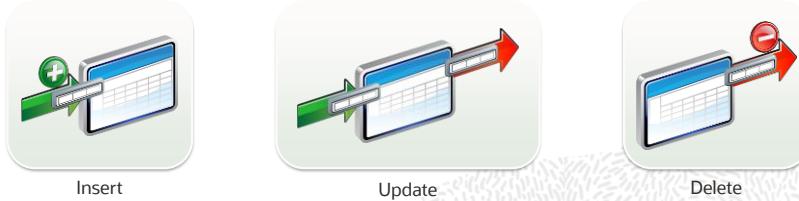


0

For Instructor Use Only.  
This document should not be distributed.

# Data Manipulation Language

- A DML statement is executed when you:
  - Add new rows to a table
  - Modify existing rows in a table
  - Remove existing rows from a table
- A transaction consists of a collection of DML statements that form a logical unit of work.



6

O

Data manipulation language (DML) is a core part of SQL. 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*.

Consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction might consist of three separate operations: decreasing the savings account, increasing the checking account, and recording the transaction in the transaction journal. The database server must guarantee that all the three SQL statements are performed to maintain the accounts in proper balance. When something prevents one of the statements in the transaction from executing, the other statements of the transaction must be undone.

**Note:** Most of the DML statements in this lesson assume that no constraints on the table are violated. Constraints are discussed later in this course.

# Adding a New Row to a Table

departments				70 Public Relations	100	1700	New row
DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID				
1	10 Administration	200	1700				
2	20 Marketing	201	1800				
3	50 Shipping	124	1500				
4	60 IT	103	1400				
5	80 Sales	149	2500				
6	90 Executive	100	1700				
7	110 Accounting	205	1700				
8	190 Contracting	(null)	1700				

Insert a new row into the departments table.

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	70 Public Relations	100	1700
2	10 Administration	200	1700
3	20 Marketing	201	1800
4	50 Shipping	124	1500
5	60 IT	103	1400
6	80 Sales	149	2500
7	90 Executive	100	1700
8	110 Accounting	205	1700
9	190 Contracting	(null)	1700

7

0

The graphic in the slide illustrates the addition of a new department record to the departments table.

# INSERT Statement Syntax

- Add new rows to a table by using the INSERT statement:

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

- Each comma-separated list of values, enclosed in parentheses, inserts a new row.
- Enclose character and date data in quotation marks.



0

8

You can add new rows to a table by issuing the `INSERT` statement.

In the syntax:

- **table:** the name of the table. The table name is required.
- **column:** the names of the columns in the table to populate. The column names are optional. As described below, the values listed depend on whether and how column names are provided.
- **value:** the corresponding value for each column. The list of values for each row are enclosed within parentheses and separated by commas. If the column names are provided, the list of values must be in the same order as the list of corresponding column names. If the column names are not provided, each list of values must include all columns in the order provided by the `DESCRIBE` statement.

# Inserting New Rows: Listing Column Names

If you include the list of column names after the table name:

- You can list the columns in any order.
- The values must be listed in the same order as the columns they populate.
- The list of values has to include the same number of values as the number of columns.

```
INSERT INTO departments(department_id,  
department_name, manager_id, location_id)  
VALUES (70, 'Public Relations', 100, 1700),  
(150, 'Shareholder Services', 100, 1700);
```

2 row(s) affected  
Records: 2 Duplicates: 0 Warnings: 0

9

The list of column names in parentheses after the table name in the `INSERT` statement is optional. If you include the list of column names, it does not have to be in the same order as the default order of the columns in the table. The list of column names can be in any order, but the values listed must be in the same order as the list of columns and has to have the same number of values as the list of columns.

The example in this slide inserts two rows. You could also create two separate `INSERT` statements, one for each row.

# Inserting New Rows: Omitting Column Names

If you omit the list of column names after the table name:

- You must include a value for every column in the table.
- The values must be listed in the same order as the default order of the columns, as displayed by a `DESCRIBE table_name;` statement.

```
INSERT INTO departments
VALUES (120, 'Treasury', 100, 1700),
       (130, 'Corporate Tax', 100, 1700),
       (140, 'Control and Credit', 100, 1700);
```

3 row(s) affected  
Records: 3 Duplicates: 0 Warnings: 0

10

0

The default order of columns is the order that columns are displayed as output from a `SELECT * FROM table_name;` query. You can find the default order by issuing a `DESCRIBE table_name;` statement.

# Inserting Rows with Null Values

- Implicit method: If you omit a column from the column list, the value is set to the default value for the column or `NULL` if the column does not have a default value.

```
INSERT INTO departments (department_id,  
                        department_name)  
VALUES      (30, 'Purchasing');
```

1 row(s) affected

- Explicit method: You can specify the `NULL` keyword in the `VALUES` list.

```
INSERT INTO departments  
VALUES      (100, 'Finance', NULL, NULL);
```

1 row(s) affected

11

0

If you include a column list, but omit one or more of the columns, the value is set to the default value for the column or `NULL`. You learn about setting default values in the lesson titled "Introduction to Data Definition Language in MySQL." Be sure that you can use null values in the targeted column by verifying the `NULL` status with the `DESCRIBE` command.

# Inserting Special Values in MySQL

The CURDATE() function records the current date and time in MySQL.

```
INSERT INTO employees (employee_id,
                      first_name, last_name,
                      email, phone_number,
                      hire_date, job_id, salary,
                      commission_pct, manager_id,
                      department_id)
VALUES
(113,
'Louis', 'Popp',
'LPOPP', '515.124.4567',
CURDATE(), 'AC_ACCOUNT', 6900,
NULL, 205, 110);
```

1 row(s) affected

12

You can use functions to enter special values in your table.

The example in the slide records information for employee Popp in the employees table. It supplies the current date and time in the hire\_date column. It uses the CURDATE() function that returns the current date.

## Confirming Additions to the Table

```
SELECT employee_id, last_name, job_id, hire_date, commission_pct
FROM   employees
WHERE  employee_id = 113;
```

# Inserting Specific Date and Time Values in MySQL

- Add a new employee.

```
INSERT INTO employees
VALUES
(115,
'Alexander', 'Khoo',
'AKHOO', '515.127.4562',
2011-05-18,
'SA REP', 3100, 0.2, 100, 60);
```

1 row(s) affected

- Verify your addition.

```
SELECT * FROM employees WHERE employee_id = 115;
```

#	employee_id	first_name	last_name	email	phone_number	hire_date	job_id	salary	commission_pct	manager_id	department_id
1	115	Alexander	Khoo	AKHOO	515.127.4562	2011-05-18	SA REP	3100.00	0.20	100	60
*	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL

13

0

You can enter dates using the default date format, just as you do in WHERE conditions or other references to dates.

# Inserting and Reformatting Specific Date and Time Values in MySQL

- Add a new employee.

```
INSERT INTO employees
VALUES
(114,
'Den', 'Raphealy',
'DRAPHEAL', '515.127.4561',
STR_TO_DATE('FEB 3, 2016', '%M %d, %Y'),
'SA_REP', 11000, 0.2, 100, 70);
```

1 row(s) affected

- Verify your addition.

```
SELECT * FROM employees WHERE employee_id = 114;
```

#	employee_id	first_name	last_name	email	phone_number	hire_date	job_id	salary	commission_pct	manager_id	department_id
1	114	Den	Raphealy	DRAPEAL	515.127.4561	2016-02-03	SA_REP	11000.00	0.20	100	70
*											

14

In MySQL, If you want to enter the date in a format other than the default format, you must use the `STR_TO_DATE` function.

The example in the slide records information for employee Raphealy in the `employees` table. It sets the `hire_date` column to be February 3, 2016, but stores and displays it in the default date format.

# Copying Rows from Another Table

- Write your `INSERT` statement with a subquery:

```
INSERT INTO copy_emp  
SELECT * FROM employees;
```

23 row(s) affected  
Records: 23 Duplicates: 0 Warnings: 0

```
INSERT INTO sales_reps(id, name, salary, commission_pct)  
SELECT employee_id, last_name, salary, commission_pct  
FROM employees WHERE job_id LIKE '%REP%';
```

6 row(s) affected  
Records: 6 Duplicates: 0 Warnings: 0

- Do not use the `VALUES` clause.
- Match the number of columns in the `INSERT` clause to those in the subquery.
- It inserts all the rows returned by the subquery into the table.

15

O

You can use the `INSERT` statement to add rows to a table where the values are derived from existing tables. In the example in the slide, for the `INSERT INTO` statement to work, you must have already created the `sales_reps` table using the `CREATE TABLE` statement. `CREATE TABLE` is discussed in the lesson titled “Introduction to Data Definition Language.”

In place of the `VALUES` clause, you use a subquery.

## Syntax

```
INSERT INTO table [ column (, column) ] subquery;
```

In the syntax:

<code>table</code>	Is the name of the table
<code>column</code>	Is the name of the column in the table to populate
<code>subquery</code>	Is the subquery that returns rows to the table

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. Zero or more rows are added depending on the number of rows returned by the subquery.

# Lesson Agenda

- Adding new rows in a table
  - `INSERT` statement
- Changing data in a table
  - `UPDATE` statement
- Removing rows from a table:
  - `DELETE` statement
  - `TRUNCATE` statement
- Database transaction control using `COMMIT`, `ROLLBACK`, and `SAVEPOINT`
- Read consistency
- Manual Data Locking
  - `FOR UPDATE` clause in a `SELECT` statement
  - `LOCK TABLE` statement

16

0



For Instructor Use Only.  
This document should not be distributed.

# Changing Data in a Table

employees

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	MANAGER_ID	COMMISSION_PCT	DEPARTMENT_ID
100 Steven	King	King	24000	(null)	(null)	90
101 Neena	Kochhar	Kochhar	17000	100	(null)	90
102 Lex	De Haan	De Haan	17000	100	(null)	90
103 Alexander	Hunold	Hunold	9000	102	(null)	60
104 Bruce	Ernst	Ernst	6000	103	(null)	60
107 Diana	Lorentz	Lorentz	4200	103	(null)	60
124 Kevin	Mourgos	Mourgos	5800	100	(null)	50

Update rows in the employees table:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	MANAGER_ID	COMMISSION_PCT	DEPARTMENT_ID
100 Steven	King	King	24000	(null)	(null)	90
101 Neena	Kochhar	Kochhar	17000	100	(null)	90
102 Lex	De Haan	De Haan	17000	100	(null)	90
103 Alexander	Hunold	Hunold	9000	102	(null)	80
104 Bruce	Ernst	Ernst	6000	103	(null)	80
107 Diana	Lorentz	Lorentz	4200	103	(null)	80
124 Kevin	Mourgos	Mourgos	5800	100	(null)	50

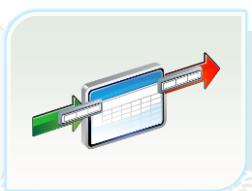
The slide illustrates changing the department number for employees in department 60 to department 80.

# UPDATE Statement Syntax

- Modify existing values in a table with the UPDATE statement:

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

- Update more than one row at a time (if required).



O

18

You can modify the existing values in a table by using the UPDATE statement.

In the syntax:

<i>table</i>	Is the name of the table
<i>column</i>	Is the name of the column in the table to populate
<i>value</i>	Is the corresponding value or subquery for the column
<i>Condition</i>	Identifies the rows to be updated and is composed of column names, expressions, constants, subqueries, and comparison operators

Confirm the update operation by querying the table to display the updated rows.

For more information, see the section on “UPDATE” in *Oracle Database SQL Language Reference* for 19c database.

**Note:** In general, use the primary key column in the WHERE clause to identify a single row for update. Using other columns can unexpectedly cause several rows to be updated. For example, identifying a single row in the employees table by last name may return more than one employee having the same last name.

# Updating Rows in a Table

- Values for a specific row or rows are modified if you specify the WHERE clause:

```
UPDATE employees
SET department_id = 50
WHERE employee_id = 113;
```

```
1 row(s) affected
Rows matched: 1 Changed: 1 Warnings: 0
```

- Values for all the rows in the table are modified if you omit the WHERE clause:

```
UPDATE copy_emp
SET department_id = 110;
```

```
20 row(s) affected
Rows matched: 23 Changed: 20 Warnings: 0
```

- Specify SET column\_name= NULL to update a column value to NULL.

19

O

The UPDATE statement modifies the values of a specific row or rows if the WHERE clause is specified. The example in the slide shows the transfer of employee 113 (Popp) to department 50.

If you omit the WHERE clause, values for all the rows in the table are modified. Examine the updated rows in the COPY\_EMP table.

```
SELECT last_name, department_id
FROM copy_emp;
```

For example, an employee who was an SA REP has now changed his job to an IT PROG. Therefore, his JOB\_ID needs to be updated and the commission field needs to be set to NULL.

```
UPDATE employees
SET job_id = 'IT_PROG', commission_pct = NULL
WHERE employee_id = 114;
```

**Note:** The COPY\_EMP table has the same data as the EMPLOYEES table.

# Updating Rows Based on Another Table

Use the subqueries in the UPDATE statements to update row values in a table based on values from another table:

```
UPDATE copy_emp
SET department_id = (SELECT department_id
                      FROM employees
                      WHERE employee_id = 100)
WHERE job_id      = (SELECT job_id
                      FROM employees
                      WHERE employee_id = 200);
```

1 row(s) affected  
Rows matched: 1 Changed: 1 Warnings: 0

You can use the subqueries in the UPDATE statements to update values in a table. The example in the slide updates the `copy_emp` table based on the values from the `employees` table. It changes the department number to employee 100's current department number (90) for all the employees whose job ID is the same as employee 200's job ID. In MySQL you cannot update a table and select from the same table in a subquery.



# Quiz

In an UPDATE statement, you can use a subquery for which of the following clauses?

- a. The SET clause only.
- b. The WHERE clause only.
- c. Either the SET clause or the WHERE clause or both clauses.
- d. Neither the SET clause or the WHERE clause.

21

0

**Answer: c**

For Instructor Use Only.  
This document should not be distributed.

# Lesson Agenda

- Adding new rows in a table
  - INSERT statement
- Changing data in a table
  - UPDATE statement
- Removing rows from a table:
  - DELETE statement
  - TRUNCATE statement
- Database transaction control using COMMIT, ROLLBACK, and SAVEPOINT
- Read consistency
- Manual Data Locking
  - FOR UPDATE clause in a SELECT statement
  - LOCK TABLE statement

22

0



For Instructor Use Only.  
This document should not be distributed.

# Removing a Row from a Table

departments

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700
8	190	Contracting	(null)	1700

Delete a row from the departments table:

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700

23

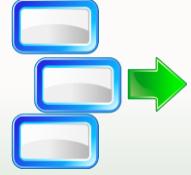
0

The contracting department has been removed from the departments table (assuming no constraints on the departments table are violated), as shown in the graphic in the slide.

# DELETE Statement

You can remove existing rows from a table by using the **DELETE** statement:

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



0

24

In the syntax:

*table* Is the name of the table

*condition* Identifies the rows to be deleted, and is composed of column names, expressions, constants, subqueries, and comparison operators

**Note:** If no rows are deleted, the message “0 rows deleted” is returned (on the Script Output tab in SQL Developer).

For more information, see the section on “**DELETE**” in *Oracle Database SQL Language Reference* for 19c database.

# Deleting Rows from a Table

- Specific rows are deleted if you specify the WHERE clause:

```
DELETE FROM departments  
WHERE department_name = 'Finance';
```

1 row(s) affected

- All rows in the table are deleted if you omit the WHERE clause:

```
DELETE FROM copy_emp;
```

23 row(s) affected

25

0

You can delete specific rows by specifying the WHERE clause in the DELETE statement. The first example in the slide deletes the accounting department from the DEPARTMENTS table. You can confirm the delete operation by trying to display the deleted rows using the SELECT statement. The query returns 0 rows.

```
SELECT *  
FROM departments  
WHERE department_name = 'Finance';
```

However, if you omit the WHERE clause, all rows in the table are deleted. The second example in the slide deletes all rows from the COPY\_EMP table, because no WHERE clause was specified.

## Example

Remove rows identified in the WHERE clause.

```
DELETE FROM employees WHERE employee_id = 115;
```

```
DELETE FROM departments WHERE department_id IN (130, 140);
```

# Deleting Rows Based on Another Table

Use the subqueries in the `DELETE` statements to remove rows from a table based on values from another table:

```
DELETE FROM employees
WHERE department_id IN
  (SELECT department_id
   FROM departments
   WHERE department_name
     LIKE '%Public%');
```

1 row(s) affected

You can use the subqueries to delete rows from a table based on values from another table. The example in the slide deletes all the employees in a department, where the department name contains the string `Public`.

The subquery searches the `departments` table to find the department number based on the department name containing the string `Public`. The subquery then feeds the department number to the main query, which deletes rows of data from the `employees` table.

# TRUNCATE Statement

- Removes all rows from a table, leaving the table empty and the table structure intact
- Is a data definition language (DDL) statement rather than a DML statement; cannot be undone
- Syntax:

```
TRUNCATE TABLE table_name;
```

- Example:

```
TRUNCATE TABLE copy_emp;
```

You can use the TRUNCATE statement to quickly remove all rows from a table or cluster efficiently. Removing rows with the TRUNCATE statement is faster than removing them with the DELETE statement for the following reasons:

- The TRUNCATE statement is a data definition language (DDL) statement and generates no rollback information. Rollback information is covered later in this lesson.

If the table is the parent of a referential integrity constraint, you cannot truncate the table. You need to disable the constraint before issuing the TRUNCATE statement. You will learn more about DDL statements and disabling constraints in the lesson titled “Introduction to Data Definition Language.”

# Lesson Agenda

- Adding new rows in a table
  - INSERT statement
- Changing data in a table
  - UPDATE statement
- Removing rows from a table:
  - DELETE statement
  - TRUNCATE statement
- Database transaction control using COMMIT, ROLLBACK, and SAVEPOINT
- Read consistency
- Manual Data Locking
  - FOR UPDATE clause in a SELECT statement
  - LOCK TABLE statement



0

For Instructor Use Only.  
This document should not be distributed.

# Multiple-statement Transactions

- A collection of statements that are treated as a single unit.
  - Groups multiple statements
  - Is useful when multiple clients can access data from the same table at the same time
  - Is important when statements affect the same or related data
- All or none of the steps succeed.
  - Execute if all steps are successful.
  - Cancel if any steps cause errors or do not complete.



O

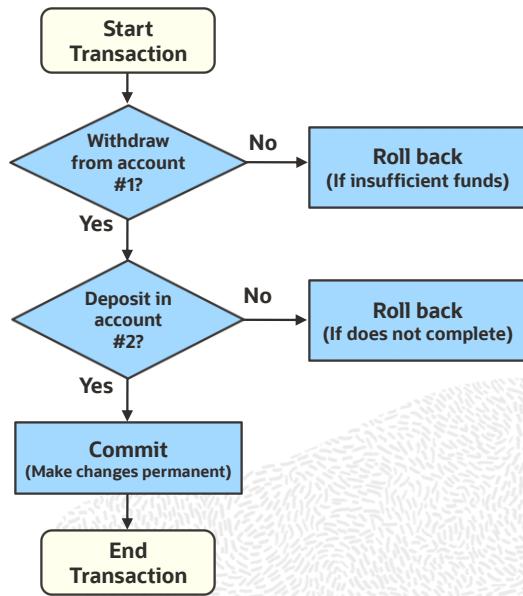
29

A transaction lets you execute one or more SQL statements as a single unit of work, so that either all or none of the statements succeed. This happens independently of work being done by any other transactions. If all the statements succeed, you commit the transaction to record the effect permanently in the database. If an error occurs during the transaction, you roll it back to cancel it. Any statements executed up to the point of canceling within the transaction are undone, leaving the database in the state it was in before the transaction.

**Note:** MySQL supports transactions only for tables that use a transactional storage engine (such as InnoDB). These statements have no noticeable effect on tables managed by non-transactional storage engines.

# Transaction Diagram

Example of a banking transaction:



30

O

The diagram in the slide shows an attempted transfer of \$1,000 from your savings account to your checking account. You would not be happy if the money were successfully withdrawn from your savings account, but never reached your checking account.

To protect against this kind of error, the program that handles your transfer request begins a transaction, and then issues the SQL statements needed to move the money from your savings to your checking account. It only commits the transaction if everything succeeds. If a problem occurs, the program instructs the server to undo all the changes made since the transaction began.

# AUTOCOMMIT and Transaction Control Statements

By default, MySQL runs with the `AUTOCOMMIT` mode enabled. When `AUTOCOMMIT` mode is enabled:

- As soon as you execute a statement that modifies a table, MySQL stores the update on disk.
- Each statement is treated like a transaction.

Use the following statements to control a multiple-statement transaction:

- `START TRANSACTION` (or `BEGIN`): Explicitly begins a new transaction and disables the `AUTOCOMMIT` mode
- `COMMIT`: Makes the changes from the current transaction permanent
- `ROLLBACK`: Cancels the changes from the current transaction

When you issue a `START TRANSACTION` statement, `AUTOCOMMIT` mode is disabled. Results of statements after the `START TRANSACTION` statement that modify a table are not written to the disk. The transaction remains open until you close it with either a `COMMIT`, a `ROLLBACK`, or any statement that implicitly closes a transaction.

You can disable `AUTOCOMMIT`. After disabling `AUTOCOMMIT` mode, changes to transaction-safe tables (such as InnoDB) are not made permanent immediately. You must use a `COMMIT` statement to store your changes to disk or `ROLLBACK` statement to ignore the changes. The `AUTOCOMMIT` mode then reverts to its previous state.

To disable `AUTOCOMMIT`, enter the following statement:

```
SET AUTOCOMMIT=0;
```

# Committing Data in a Transaction

- Make the changes to the data and commit the transaction:

```
START TRANSACTION;

INSERT INTO retired_employees (employee_id, first_name, last_name, email, retired_date, job_id, salary, manager_id,
department_id)
SELECT employee_id, first_name, last_name, email, CURDATE(), job_id, salary, manager_id, department_id
FROM employees
WHERE employee_id = 113;

DELETE from employees where employee_id = 113;

COMMIT;
```

32

0

In the example in the slide, an employee is being moved from the `employees` table to the `retired_employees` table. Making this a transaction ensures that the employee can be added to the `retired_employees` table before being deleted from the `employees` table. If either of these actions fails, the transaction is rolled back. Because both statements execute correctly, the changes are saved by issuing the `COMMIT` statement. In this example, the columns to be inserted must be listed because not all of the columns from the `employees` table are used in the `retired_employees` table. Also the `retired_employees` table has a `retired_date` column, which is set to the current date in this example, but could be set to a specified date if desired.

# Rolling Back Changes

- Make changes to data and then roll back the changes:

```
START TRANSACTION;

INSERT INTO departments
VALUES (200, 'Operations', NULL, 1700);

UPDATE employees
SET department_id = 200
WHERE department_id =
(SELECT department_id FROM departments WHERE department_name = 'IT');

ROLLBACK;
```

33

0

In the example in the slide, employees from the IT department are being transferred to the newly created Operations department (200). Creating a transaction ensures that the new Operations department is successfully added to the departments table before employees from the IT department have their department ID set to the new department. In this example, the ROLLBACK statement returns the departments table and the employees table back to the status they were in before the transaction began. That is, the department that was inserted is removed, and the employees from IT still have their previous department ID.

# Rolling Back Changes to a Marker

- Create a marker in the current transaction by using the `SAVEPOINT` statement.
- Roll back to that marker by using the `ROLLBACK TO SAVEPOINT` statement.

```
START TRANSACTION;

INSERT INTO copy_emp
VALUES (113, 'Luis', 'Popp', 'LPOPP', '515.124.4567',
       STR_TO_DATE('07-12-2015', '%d-%m-%Y'), 'FI_ACCOUNT', 6900, NULL,
       205, 110);

SAVEPOINT first_insert;

INSERT INTO copy_emp
VALUES (175, 'Alyssa', 'Hutton', 'AHUTTON', '011.44.1644.429266',
       STR_TO_DATE('19-03-2013', '%d-%m-%Y'), 'SA REP', 8800, .25, 149,
       80);

ROLLBACK TO first_insert;

COMMIT;
```

34

O

You can create a marker in the current transaction by using the `SAVEPOINT` statement, which divides the transaction into smaller sections. You can then discard pending changes back to that marker by using the `ROLLBACK TO SAVEPOINT` statement. The effect of the example in this slide is that the first `INSERT` statement is committed, but the second insert statement is rolled back. That is employee 113 is inserted, but employee 175 is not.

Note that if you create a second savepoint with the same name as an earlier savepoint, the earlier savepoint is deleted.

The following transaction would commit the first and third `INSERT` statements (employees 177 and 179) but not the second one (employee 178):

```
START TRANSACTION;

INSERT INTO copy_emp
VALUES (177, 'Jack', 'Livingston', 'JLIVINGS', '011.44.1644.429264',
       STR_TO_DATE('23-04-2014', '%d-%m-%Y'), 'SA REP', 8400, .20, 149, 80);

SAVEPOINT first_insert;

INSERT INTO copy_emp
VALUES (178, 'Kimberely', 'Grant', 'KGRANT', '011.44.1644.429263',
       STR_TO_DATE('24-05-2015', '%d-%m-%Y'), 'SA REP', 7000, .15, 149, NULL);

ROLLBACK TO first_insert;

INSERT INTO copy_emp
VALUES (179, 'Charles', 'Johnson', 'CJOHNSON', '011.44.1644.429262',
       STR_TO_DATE('04-01-2016', '%d-%m-%Y'), 'SA REP', 6200, .10, 149, 80);

COMMIT;
```

# Lesson Agenda

- Adding new rows in a table
  - INSERT statement
- Changing data in a table
  - UPDATE statement
- Removing rows from a table:
  - DELETE statement
  - TRUNCATE statement
- Database transaction control using COMMIT, ROLLBACK, and SAVEPOINT
- Consistent reads
- Manual Data Locking
  - FOR UPDATE clause in a SELECT statement
  - LOCK TABLE statement



0

For Instructor Use Only.  
This document should not be distributed.

# Consistent Reads

In busy database systems, some transactions are reading data while other transactions are changing that same data.

- A read operation does not place any locks on the tables it is reading. Other transactions are not blocked from modifying the data.
- A read operation reads a snapshot of the data. Other transactions can then modify the data.
- The timing of the snapshot depends on the isolation level.
- The default isolation level for the MySQL InnoDB storage engine is `REPEATABLE READ`. With that isolation level, a snapshot is taken with the first read statement of a transaction, and all other reads within that transaction use the same snapshot.

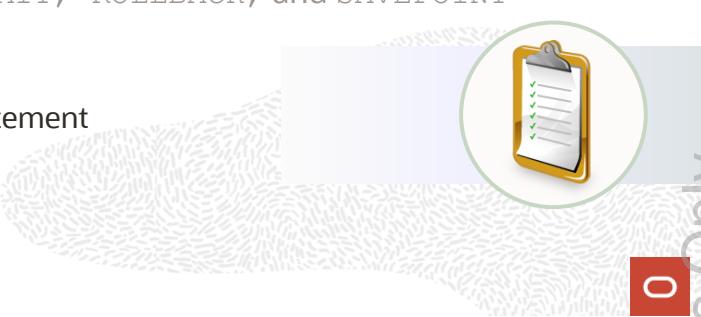
MySQL supports other isolation levels. Those are covered in the MySQL Performance Tuning course.

# Lesson Agenda

- Adding new rows in a table
  - INSERT statement
- Changing data in a table
  - UPDATE statement
- Removing rows from a table:
  - DELETE statement
  - TRUNCATE statement
- Database transaction control using COMMIT, ROLLBACK, and SAVEPOINT
- Read consistency
- Manual Data Locking
  - FOR UPDATE clause in a SELECT statement
  - LOCK TABLE statement

37

0



For Instructor Use Only.  
This document should not be distributed.

# FOR UPDATE Clause in a SELECT Statement

- Locks the rows in the EMPLOYEES table where job\_id is SA\_REP.

```
SELECT employee_id, salary, commission_pct, job_id  
FROM employees  
WHERE job_id = 'SA_REP'  
ORDER BY employee_id  
FOR UPDATE;
```

- Only after START TRANSACTION or with AUTOCOMMIT set to 0.
- Lock is released only when you issue a ROLLBACK or a COMMIT.
- If the SELECT statement attempts to lock a row that is locked by another user, the database waits until the row is available, and then returns the results of the SELECT statement.
- In MySQL, the FOR UPDATE clause must be the last clause in the SELECT statement.

When you issue a SELECT statement against the database to query some records, no locks are placed on the selected rows. In general, this is required because the number of records locked at any given time is (by default) kept to the absolute minimum: only those records that have been changed but not yet committed are locked. Even then, others will be able to read those records as they appeared before the change (the “before image” of the data). There are times, however, when you may want to lock a set of records even before you change them in your program. MySQL offers the FOR UPDATE clause of the SELECT statement to perform this locking. This works only within a transaction (after a START TRANSACTION statement) or if AUTOCOMMIT is set to 0 (off) rather than the default value of 1 (ON).

When you issue a SELECT...FOR UPDATE statement in a transaction, the database automatically obtains exclusive row-level locks on all the rows identified by the SELECT statement, thereby holding the records “for your changes only.” No one else will be able to change any of these records until you perform a ROLLBACK or a COMMIT.

You can append the optional keyword NOWAIT to the FOR UPDATE clause to tell the server not to wait if the table has been locked by another user. In this case, the query returns an error, and control is returned immediately so that you can perform other work, or you can wait for a period of time before trying again. Without the NOWAIT clause, your process blocks until the table is available, when the locks are released by the other user through the other user's issue of a COMMIT or a ROLLBACK statement.

## FOR UPDATE Clause: Examples

- You can use the `FOR UPDATE` clause in a `SELECT` statement against multiple tables.

```
SELECT e.employee_id, e.last_name, e.job_id
FROM employees e JOIN departments d
USING (department_id)
WHERE d.department_name = 'Shipping'
ORDER BY e.employee_id
FOR UPDATE;
```

- Rows from both the `EMPLOYEES` and `DEPARTMENTS` tables are locked.
- Use `FOR UPDATE OF table_name` to qualify the table that you intend to change; then only the rows from that specific table are locked.

39

0

In the example in the slide, the statement locks rows in the `employees` table with `department_id` of 50 (Shipping), and locks rows in the `departments` table with `departments` named Shipping.

In MySQL, you can use the `FOR UPDATE OF table_name` to qualify the table that you intend to change. When `OF table_name` is omitted, the query locks all tables referenced by the `FROM` clause. If an alias is specified as the table name in the `SELECT` statement, the `FOR UPDATE OF` clause must use the alias.

The following statements lock only those rows in the `employees` table with department ID of 50 (Shipping). No rows are locked in the `departments` table:

```
START TRANSACTION;

SELECT e.employee_id, e.last_name, e.job_id
FROM employees e JOIN departments d
USING (department_id)
WHERE d.department_name = 'Shipping'
ORDER BY e.employee_id
FOR UPDATE OF e;

COMMIT;
```

You can manually lock and unlock tables with the `LOCK TABLES` and `UNLOCK TABLES` statements. These statements are covered in the MySQL Performance Tuning course.

# Summary

In this lesson, you should have learned how to use the following statements:

Function	Description
INSERT	Adds a new row to the table
UPDATE	Modifies existing rows in the table
DELETE	Removes existing rows from the table
TRUNCATE	Removes all rows from a table
COMMIT	Makes all pending changes permanent
SAVEPOINT	Is used to roll back to the savepoint marker
ROLLBACK	Discards all pending data changes
FOR UPDATE clause in SELECT	Locks rows identified by the SELECT query

40

For Instructor Use Only.  
This document should not be distributed.

In this lesson, you should have learned how to manipulate data in the database by using the **INSERT**, **UPDATE**, **DELETE**, and **TRUNCATE** statements, as well as how to control data changes by using the **COMMIT**, **SAVEPOINT**, and **ROLLBACK** statements. You should have also learned how to use the **FOR UPDATE clause of the SELECT statement** to lock rows for your changes only.

Remember that the server guarantees a consistent view of data at all times.

## Practice 10b: Overview

This practice covers the following topics:

- Inserting rows into the tables
- Updating and deleting rows in the table
- Controlling transactions



0

41

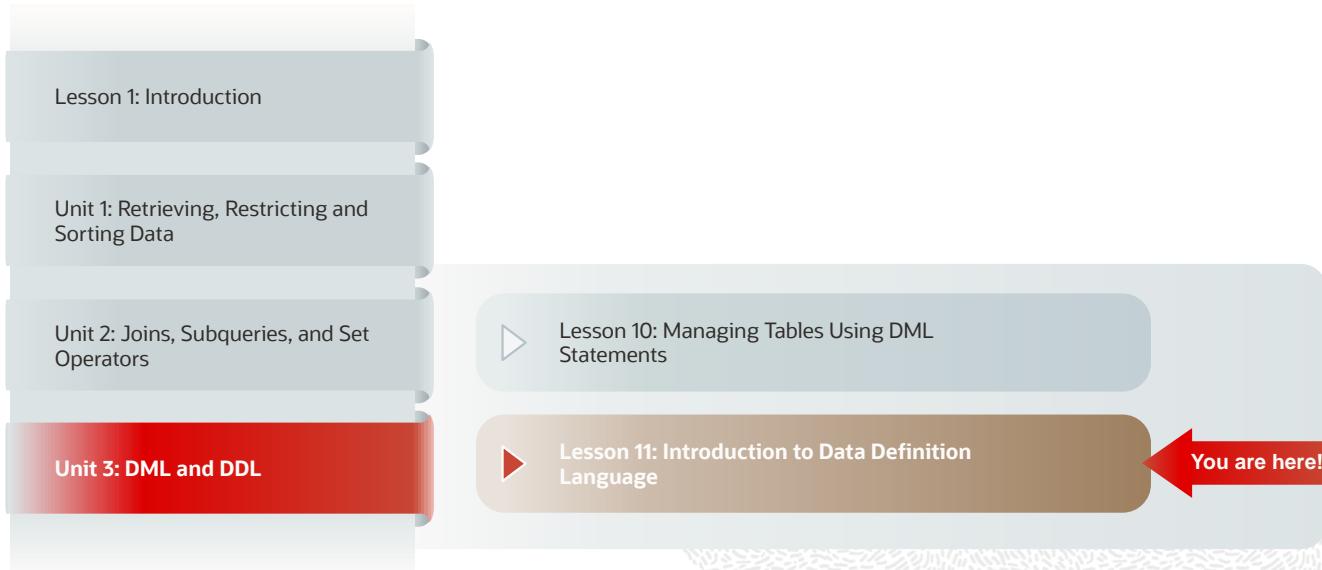
In this practice, you add rows to the `my_employee` table, update and delete data from the table, and control your transactions.

For Instructor Use Only.  
This document should not be distributed.

For Instructor Use Only.  
This document should not be distributed.

## Introduction to Data Definition Language in Oracle

# Course Roadmap



2

In Unit 3, you learn how to create and manage database objects using data definition language (DDL) statements. You will also learn how to manage data in the tables using data manipulation language (DML) statements.

# Objectives

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

- Categorize the main database objects
- Review the table structure
- List the data types that are available for columns
- Create a simple table
- Explain how constraints are created at the time of table creation

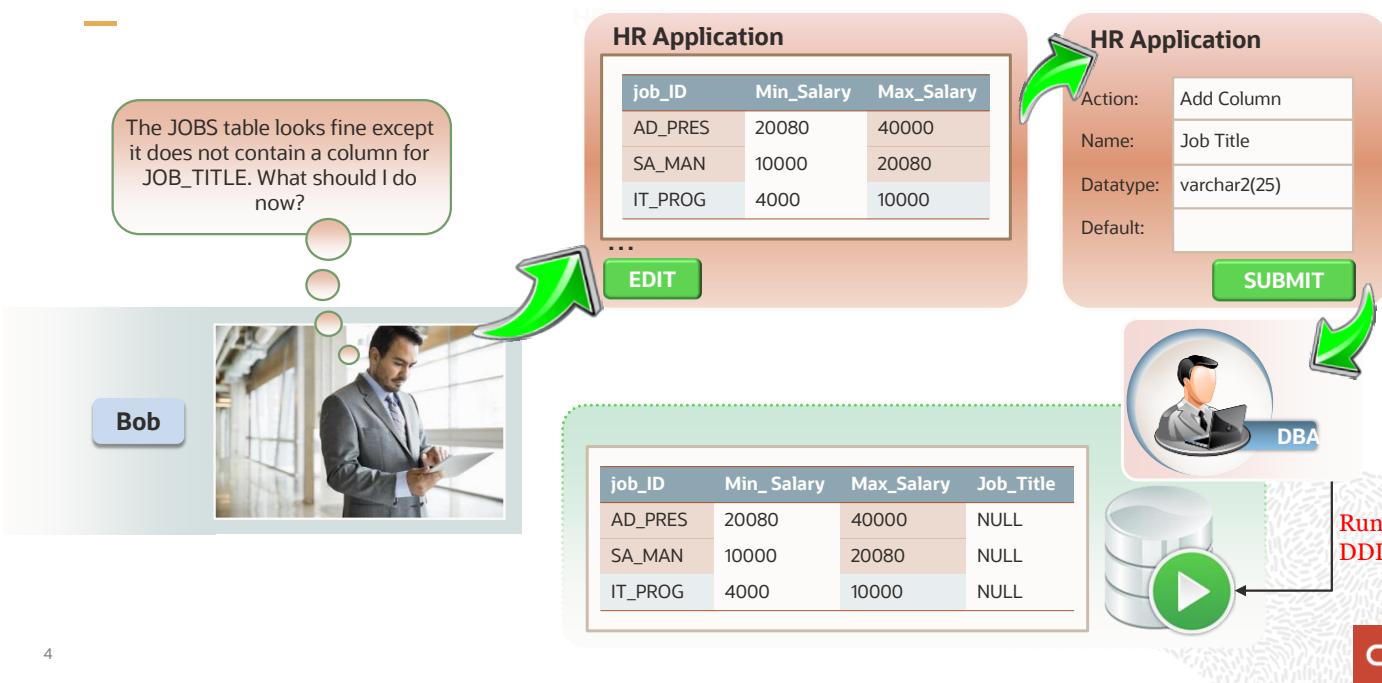


0

3

In this lesson, you are introduced to the data definition language (DDL) statements. You learn the basics of creating simple tables, altering them, and removing them. The data types available in DDL are shown and schema concepts are introduced. Constraints are discussed in this lesson. Exception messages that are generated from violating constraints during DML operations are shown and explained.

# HR Application Scenario



Consider a scenario where Bob, an HR manager is creating tables to store all the employee information in the database. While creating the `JOBS` table to store all the job information, he forgets to create a column for the job title. So what should Bob do now?

Should he drop the table and create `JOBS` table again? No!

Bob can alter the table and add a new column (`JOB_TITLE`) to the existing `JOBS` table. The statements that allow you to modify the structure of database objects are called data definition language (DDL) statements. Usually, the permission to execute DDL statements is given only to the admin.

Bob submits the request to alter the `JOBS` table structure along with the details. The DBA receives the request and constructs an appropriate SQL statement to alter the `JOBS` table.

A new column called `Job_Title` is added to the `JOBS` table. The value for `Job_Title` remains `NULL` until Bob goes to the application and updates the values for all the records.

In this lesson, you learn more about DDL statements.

# Lesson Agenda

- Database objects
  - Naming rules
- CREATE TABLE statement
- Data types
- Overview of constraints: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK constraints
- Creating a table using a subquery
- ALTER TABLE statement
- DROP TABLE statement

5

0



For Instructor Use Only.  
This document should not be distributed.

# Database Objects

Object	Description
Table	Is the 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 some queries
Synonym	Gives alternative name to an object



6

0

The Oracle database can contain multiple data objects. Remember to outline each object in the database design so that it can be created during the build stage of database development.

- **Table:** Stores data
- **View:** Is a subset of data from one or more tables
- **Sequence:** Generates numeric values
- **Index:** Improves the performance of some queries
- **Synonym:** Gives an alternative name to an object

## Oracle Table Structures

- You can create tables at any time, even when users are using the database.
- You do not need to specify the size of a table. The size is ultimately defined by the amount of space allocated to the database as a whole. It is important, however, to estimate how much space a table will use over time.
- You can also modify the table structure online.

# Naming Rules for Tables and Columns

Ensure that the table names and column names:

- Begin with a letter
- Are 1–30 characters long
- Contain only A–Z, a–z, 0–9, \_, \$, and #
- Do not duplicate the name of another object owned by the same user
- Are not Oracle server–reserved words



7

O

Name the database tables and columns according to the standard rules for naming any Oracle database object.

- Table names and column names must begin with a letter and be 1–30 characters long.
- Names must contain only the characters A–Z, a–z, 0–9, \_ (underscore), \$, and # (legal characters, but their use is discouraged).
- Names must not duplicate the name of another object owned by the same Oracle server user.
- Names must not be an Oracle server–reserved word.
  - You may also use quoted identifiers to represent the name of an object. A quoted identifier begins and ends with double quotation marks (""). If you name a schema object using a quoted identifier, you must use the double quotation marks whenever you refer to that object. Quoted identifiers can be reserved words, although this is not recommended.

## Naming Guidelines

Use descriptive names for tables and other database objects.

**Note:** Names are not case-sensitive. For example, EMPLOYEES is treated to be the same name as eMPLOYEES or eMpLOYEES. However, quoted identifiers are case-sensitive.

For more information, see the “Schema Object Names and Qualifiers” section in the *Oracle Database SQL Language Reference* for 19c database.

# Lesson Agenda

- Database objects
  - Naming rules
- CREATE TABLE statement
- Data types
- Overview of constraints: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK constraints
- Creating a table using a subquery
- ALTER TABLE statement
- DROP TABLE statement

0



For Instructor Use Only.  
This document should not be distributed.

# CREATE TABLE Statement

- You must have:
  - The CREATE TABLE privilege
  - A storage area

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

- You specify:
  - The table name
  - The column name, column data type, and column size



O

9

You create tables to store data by executing the SQL CREATE TABLE statement. This statement is one of the DDL statements that are a subset of the SQL statements used to create, modify, or remove Oracle Database structures. These statements have an immediate effect on the database and they also record information in the data dictionary. The data dictionary is an important set of read-only tables that provide database information.

To create a table, a user must have the CREATE TABLE privilege and a storage area in which to create objects. The database administrator (DBA) uses data control language (DCL) statements to grant privileges to users.

In the syntax:

<i>schema</i>	Is the same as the owner's name
<i>table</i>	Is the name of the table
<i>DEFAULT expr</i>	Specifies a default value if a value is omitted in the INSERT statement
<i>column</i>	Is the name of the column
<i>datatype</i>	Is the column's data type and length

**Note:** The CREATE ANY TABLE privilege is needed to create a table in any schema other than the user's schema.

# Creating Tables

- Create the table:

```
CREATE TABLE dept
  (deptno      NUMBER(2),
   dname       VARCHAR2(14),
   loc         VARCHAR2(13),
   create_date DATE DEFAULT SYSDATE);
table DEPT created.
```

- Confirm table creation:

```
DESCRIBE dept
```

DESCRIBE dept		
Name	Null	Type
DEPTNO		NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)
CREATE_DATE		DATE

10

O

The example in the slide creates the `DEPT` table with four columns: `DEPTNO`, `DNAME`, `LOC`, and `CREATE_DATE`.

The `CREATE_DATE` column has a default value. If a value is not provided for an `INSERT` statement, the system date is automatically inserted.

To confirm that the table was created, run the `DESCRIBE` command.

Because creating a table is a DDL statement, an automatic commit takes place when this statement is executed.

**Note:** You can view the list of tables that you own by querying the data dictionary, as shown in the following example:

```
select table_name from user_tables;
```

Using data dictionary views, you can also find information about other database objects, such as views, indexes, and so on. You will learn about data dictionaries in detail in Lesson 19 the course.

# Lesson Agenda

- Database objects
  - Naming rules
- CREATE TABLE statement
- Data types
- Overview of constraints: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK constraints
- Creating a table using a subquery
- ALTER TABLE statement
- DROP TABLE statement

0



For Instructor Use Only.  
This document should not be distributed.

# Data Types

Data Type	Description
<code>VARCHAR2 (size)</code>	Variable-length character data
<code>CHAR (size)</code>	Fixed-length character data
<code>NUMBER (p, s)</code>	Variable-length numeric data
<code>DATE</code>	Date and time values
<code>LONG</code>	Variable-length character data (up to 2 GB)
<code>CLOB</code>	Maximum size is $(4 \text{ gigabytes} - 1) * (\text{DB\_BLOCK\_SIZE})$ .
<code>RAW and LONG RAW</code>	Raw binary data
<code>BLOB</code>	Maximum size is $(4 \text{ gigabytes} - 1) * (\text{DB\_BLOCK\_SIZE})$ initialization parameter (8 TB to 128 TB).
<code>BFILE</code>	Binary data stored in an external file (up to 4 GB)
<code>ROWID</code>	A base-64 number system representing the unique address of a row in its table

12

When you identify a column for a table, you need to provide a data type for the column. There are several data types available:

Data Type	Description
<code>VARCHAR2 (size)</code>	Variable-length character data (A maximum <code>size</code> must be specified; minimum <code>size</code> is 1.) Maximum size is: <ul style="list-style-type: none"> <li>• 32767 bytes if <code>MAX_SQL_STRING_SIZE = EXTENDED</code></li> <li>• 4000 bytes if <code>MAX_SQL_STRING_SIZE = LEGACY</code></li> </ul>
<code>CHAR [ (size) ]</code>	Fixed-length character data of length <code>size</code> bytes (Default and minimum <code>size</code> is 1; maximum <code>size</code> is 2,000.)
<code>NUMBER [ (p, s) ]</code>	Number having precision <code>p</code> and scale <code>s</code> (Precision is the total number of decimal digits and scale is the number of digits to the right of the decimal point; precision can range from 1 through 38, and scale can range from -84 through 127.)
<code>DATE</code>	Date and time values to the nearest second between January 1, 4712 B.C., and December 31, 9999 A.D.

Data Type	Description
LONG	Variable-length character data (up to 2 GB)
CLOB	A character large object containing single-byte or multibyte characters. Maximum size is $(4 \text{ gigabytes} - 1) * (\text{DB\_BLOCK\_SIZE})$ ; stores national character set data.
NCLOB	A character large object containing Unicode characters. Both fixed-width and variable-width character sets are supported, both using the database national character set. Maximum size is $(4 \text{ gigabytes} - 1) * (\text{database block size})$ ; stores national character set data.
RAW (size)	Raw binary data of length <code>size</code> bytes. You must specify <code>size</code> for a <code>RAW</code> value. Maximum <code>size</code> is: 32767 bytes if <code>MAX_SQL_STRING_SIZE = EXTENDED</code> 4000 bytes if <code>MAX_SQL_STRING_SIZE = LEGACY</code>
LONG RAW	Raw binary data of variable length up to 2 gigabytes
BLOB	A binary large object. Maximum size is $(4 \text{ gigabytes} - 1) * (\text{DB\_BLOCK\_SIZE}$ initialization parameter (8 TB to 128 TB)).
BFILE	Binary data stored in an external file (up to 4 GB)
ROWID	Base 64 string representing the unique address of a row in its table. This data type is primarily for values returned by the <code>ROWID</code> pseudocolumn.

## Guidelines

- A `LONG` column is not copied when a table is created using a subquery.
- A `LONG` column cannot be included in a `GROUP BY` or an `ORDER BY` clause.
- Only one `LONG` column can be used per table.
- No constraints can be defined on a `LONG` column.
- You might want to use a `CLOB` column rather than a `LONG` column.

# Datetime Data Types

You can use several datetime data types:

Data Type	Description
<code>TIMESTAMP</code>	Date with fractional seconds
<code>INTERVAL YEAR TO MONTH</code>	Stored as an interval of years and months
<code>INTERVAL DAY TO SECOND</code>	Stored as an interval of days, hours, minutes, and seconds



0

Data Type	Description
<code>TIMESTAMP</code>	Enables storage of time as a date with fractional seconds. It stores the year, month, day, hour, minute, and the second value of the <code>DATE</code> data type, as well as the fractional seconds value. There are several variations of this data type, such as <code>WITH TIMEZONE</code> and <code>WITH LOCALTIMEZONE</code> .
<code>INTERVAL YEAR TO MONTH</code>	Enables storage of time as an interval of years and months; used to represent the difference between two datetime values in which the only significant portions are the year and month
<code>INTERVAL DAY TO SECOND</code>	Enables storage of time as an interval of days, hours, minutes, and seconds; used to represent the precise difference between two datetime values

**Note:** The datetime data types are discussed in detail in the lesson titled “Managing Data in Different Time Zones” in the *Oracle Database: SQL Workshop II* course.

Also, for more information about datetime data types, see the sections on “`TIMESTAMP` Datatype,” “`INTERVAL YEAR TO MONTH` Datatype,” and “`INTERVAL DAY TO SECOND` Datatype” in *Oracle Database SQL Language Reference* for 19c database.

# DEFAULT Option

- Specify a default value for a column in the `CREATE TABLE` statement.

```
... hire_date DATE DEFAULT SYSDATE, ...
```

- Literal values, expressions, or SQL functions are legal values.
- Another column's name or a pseudocolumn is an illegal value.
- The default data type must match the column data type.

```
CREATE TABLE hire_dates
  (id          NUMBER(8),
   hire_date DATE DEFAULT SYSDATE);
```

```
Table HIRE_DATES created.
```

15

When you define a table, you can specify that a column should be given a default value by using the `DEFAULT` option. This option prevents null values from entering the columns when a row is inserted without a value for the column.

The default value can be a literal, an expression, or a SQL function (such as `SYSDATE` or `USER`); however, the value cannot be the name of another column or a pseudocolumn (such as `NEXTVAL` or `CURRVAL`). The default expression must match the data type of the column.

Consider the following examples:

```
INSERT INTO hire_dates values(45, NULL);
```

The preceding statement will insert the null value rather than the default value.

```
INSERT INTO hire_dates(id) values(35);
```

The preceding statement will insert `SYSDATE` for the `HIRE_DATE` column.

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

# Lesson Agenda

- Database objects
  - Naming rules
- CREATE TABLE statement
- Data types
- Overview of constraints: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK constraints
- Creating a table using a subquery
- ALTER TABLE statement
- DROP TABLE statement

16

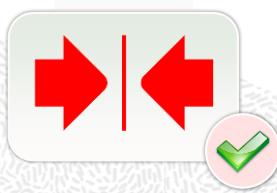
0



For Instructor Use Only.  
This document should not be distributed.

# Including Constraints

- Constraints enforce rules at the table level.
- Constraints ensure consistency and integrity of the database.
- The following constraint types are valid:
  - NOT NULL
  - UNIQUE
  - PRIMARY KEY
  - FOREIGN KEY
  - CHECK



0

17

The Oracle server uses constraints to prevent invalid data entry into tables.

You can use constraints to do the following:

- Enforce rules on the data in a table whenever a row is inserted, updated, or deleted from that table. The constraint must be satisfied for the operation to succeed.
- Prevent the dropping of a table if there are dependencies from other tables.
- Provide rules for Oracle tools, such as Oracle Developer.

## Data Integrity Constraints

Constraint	Description
NOT NULL	Specifies that the column cannot contain a null value
UNIQUE	Specifies a column or combination of columns whose values must be unique for all rows in the table
PRIMARY KEY	Uniquely identifies each row of the table
FOREIGN KEY	Establishes and enforces a referential integrity between the column and a column of the referenced table such that values in one table match values in another table
CHECK	Specifies a condition that must be true

# Constraint Guidelines

- You can name a constraint or the Oracle server generates a name by using the `SYS_Cn` format.
- Create a constraint at either of the following times:
  - At the time of table creation
  - After the creation of the table
- Define a constraint at the column or table level.
- View a constraint in the data dictionary.



18

0

All constraints are stored in the data dictionary.

Constraints are easy to reference if you give them a meaningful name. Constraint names must follow the standard object-naming rules, except that the name cannot be the same as another object owned by the same user. If you do not name your constraint, the Oracle server generates a name with the format `SYS_Cn`, where  $n$  is an integer so that the constraint name is unique.

Constraints can be defined at the time of table creation or after the creation of the table. You can define a constraint at the column or table level. Functionally, a table-level constraint is the same as a column-level constraint.

For more information, see the section on “Constraints” in *Oracle Database SQL Language Reference* for 19c database.

# Defining Constraints

- Syntax:

```
CREATE TABLE [schema.]table  
  (column datatype [DEFAULT expr]  
   [column_constraint],  
   ...  
   [table_constraint] [, . . .]);
```

- Column-level constraint syntax:

```
column [CONSTRAINT constraint_name] constraint_type,
```

- Table-level constraint syntax:

```
column,...  
[CONSTRAINT constraint_name] constraint_type  
(column, . . .),
```

The slide gives the syntax for defining constraints when creating a table. You can create constraints at the column level or the table level.

Constraints defined at the column level are included when the column is defined. Table-level constraints are defined at the end of the table definition, and must refer to the column or columns to which the constraint pertains in a set of parentheses. It is mainly the syntax that differentiates the two; otherwise, functionally, a column-level constraint is the same as a table-level constraint. NOT NULL constraints can be defined only at the column level.

Constraints that apply to more than one column must be defined at the table level.

In the syntax:

schema	Is the same as the owner's name
table	Is the name of the table
DEFAULT expr	Specifies a default value to be used if a value is omitted in the INSERT statement
column	Is the name of the column
datatype	Is the column's data type and length
column_constraint	Is an integrity constraint as part of the column definition
table_constraint	Is an integrity constraint as part of the table definition

# Defining Constraints: Example

- Example of a column-level constraint:

```
CREATE TABLE employees(
    employee_id  NUMBER(6)
        CONSTRAINT emp_emp_id_pk PRIMARY KEY,
    first_name   VARCHAR2(20),
    ...);
```

1

- Example of a table-level constraint:

```
CREATE TABLE employees(
    employee_id  NUMBER(6),
    first_name   VARCHAR2(20),
    ...
    job_id       VARCHAR2(10) NOT NULL,
    CONSTRAINT emp_emp_id_pk
        PRIMARY KEY (EMPLOYEE_ID));
```

2

20

Constraints are usually created at the same time as the table. Constraints can be added to a table after its creation and also be temporarily disabled.

Both examples in the slide create a primary key constraint on the EMPLOYEE\_ID column of the EMPLOYEES table.

1. The first example uses the column-level syntax to define the constraint.
2. The second example uses the table-level syntax to define the constraint.

More details about the primary key constraint are provided later in this lesson.

# NOT NULL Constraint

Ensures that null values are not permitted for the column:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	COMMISSION_PCT	DEPARTMENT_ID	EMAIL	PHONE_NUMBER	HIRE_DATE
100	Steven	King	24000	(null)	90	SKING	515.123.4567	17-JUN-87
101	Neena	Kochhar	17000	(null)	90	NKOCHHAR	515.123.4568	21-SEP-89
102	Lex	De Haan	17000	(null)	90	LDEHAAN	515.123.4569	13-JAN-93
103	Alexander	Hunold	9000	(null)	60	AHUNOLD	590.423.4567	03-JAN-90
104	Bruce	Ernst	6000	(null)	60	BERNST	590.423.4568	21-MAY-91
107	Diana	Lorentz	4200	(null)	60	DLORENTZ	590.423.5567	07-FEB-99
124	Kevin	Mourgos	5800	(null)	50	KMOURGOS	650.123.5234	16-NOV-99
141	Trenna	Rajs	3500	(null)	50	TRAJS	650.121.8009	17-OCT-95
142	Curtis	Davies	3100	(null)	50	CDAVIES	650.121.2994	29-JAN-97
143	Randall	Matos	2600	(null)	50	RMATOS	650.121.2874	15-MAR-98
144	Peter	Vargas	2500	(null)	50	PVARGAS	650.121.2004	09-JUL-98
149	Eleni	Zlotkey	10500	0.2	80	EZLOTKEY	011.44.1344.429018	29-JAN-00
174	Ellen	Abel	11000	0.3	80	EABEL	011.44.1644.429267	11-MAY-96
176	Jonathon	Taylor	8600	0.2	80	JTAYLOR	011.44.1644.429265	24-MAR-98
178	Kimberely	Grant	7000	0.15	(null)	KGRANT	011.44.1644.429263	24-MAY-99
200	Jennifer	Whalen	4400	(null)	10	JWHALEN	515.123.4444	17-SEP-87
201	Michael	Hartstein	13000	(null)	20	MHARTSTE	515.123.5555	17-FEB-96
202	Pat	Fay	6000	(null)	20	PFAY	603.123.6666	17-AUG-97
205	Shelley	Higgins	12000	(null)	110	SHIGGINS	515.123.8080	07-JUN-94
206	William	Gietz	8300	(null)	110	WGIEWITZ	515.123.8181	07-JUN-94

NOT NULL constraint  
(Primary Key enforces  
NOT NULL constraint.)

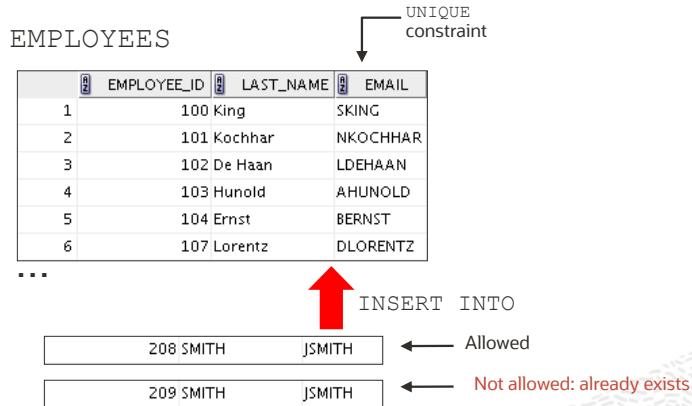
NOT NULL  
constraint

Absence of NOT NULL constraint (Any row  
can contain a null value for this column.)

The NOT NULL constraint ensures that the column contains no null values. Columns without the NOT NULL constraint can contain null values by default. NOT NULL constraints must be defined at the column level.

In the EMPLOYEES table, the EMPLOYEE\_ID column inherits a NOT NULL constraint because it is defined as a primary key. Otherwise, the LAST\_NAME, EMAIL, HIRE\_DATE, and JOB\_ID columns have the NOT NULL constraint enforced on them.

# UNIQUE Constraint



22

O

A `UNIQUE` key integrity constraint requires that every value in a column or a set of columns (key) be unique—that is, no two rows of a table can have duplicate values in a specified column or a set of columns.

The column (or set of columns) included in the definition of the `UNIQUE` key constraint is called the *unique key*. If the `UNIQUE` constraint comprises more than one column, that group of columns is called a *composite unique key*.

`UNIQUE` constraints enable the input of nulls unless you also define `NOT NULL` constraints for the same columns. In fact, any number of rows can include nulls for columns without the `NOT NULL` constraints because nulls are not considered equal to anything. A null in a column (or in all columns of a composite `UNIQUE` key) always satisfies a `UNIQUE` constraint.

**Note:** Because of the search mechanism for the `UNIQUE` constraints on more than one column, you cannot have identical values in the non-null columns of a partially null composite `UNIQUE` key.

# UNIQUE Constraint

Define at either the table level or the column level:

```
CREATE TABLE employees (
    employee_id      NUMBER(6),
    last_name        VARCHAR2(25) NOT NULL,
    email            VARCHAR2(25),
    salary           NUMBER(8,2),
    commission_pct   NUMBER(2,2),
    hire_date        DATE NOT NULL,
    ...
    CONSTRAINT emp_email_uk UNIQUE(email));
```

23

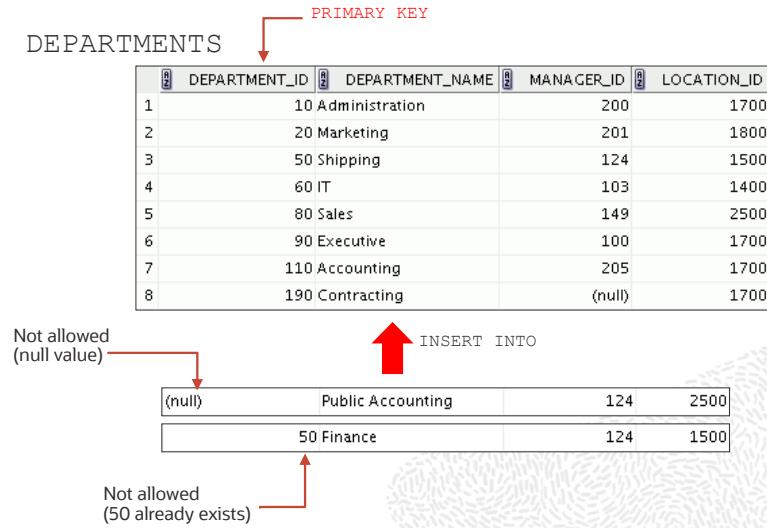
You can define UNIQUE constraints at the column level or table level.

You define the constraint at the table level when you want to create a composite unique key. A composite key is defined when there is not a single attribute that can uniquely identify a row. In that case, you can have a unique key that is composed of two or more columns, the combined value of which is always unique and can identify rows.

The example in the slide applies the UNIQUE constraint to the EMAIL column of the EMPLOYEES table. The name of the constraint is EMP\_EMAIL\_UK.

**Note:** The Oracle server enforces the UNIQUE constraint by implicitly creating a unique index on the unique key column or columns.

# PRIMARY KEY Constraint



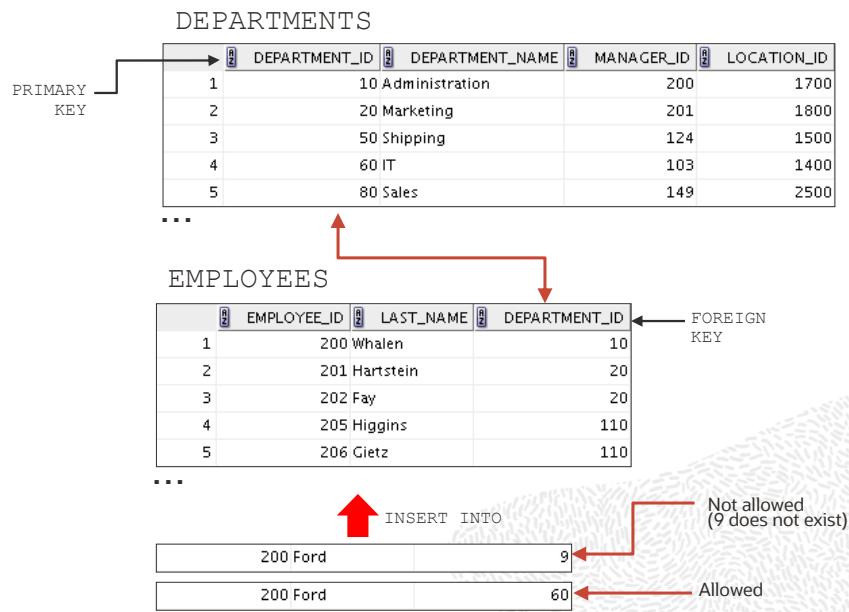
24

0

A PRIMARY KEY constraint creates a primary key for the table. You can create only one primary key for each table. The PRIMARY KEY constraint is a column or a set of columns that uniquely identifies each row in a table. This constraint enforces the uniqueness of the column or column combination, and ensures that no column can contain a null value.

**Note:** Because uniqueness is part of the primary key constraint definition, the Oracle server enforces the uniqueness by implicitly creating a unique index on the primary key column or columns.

# FOREIGN KEY Constraint



25

0

The FOREIGN KEY (or referential integrity) constraint designates a column or a combination of columns as a foreign key, and establishes a relationship with a primary key or a unique key in the same table or a different table.

In the example in the slide, DEPARTMENT\_ID has been defined as the foreign key in the EMPLOYEES table (dependent or child table); it references the DEPARTMENT\_ID column of the DEPARTMENTS table (the referenced or parent table).

## Guidelines

- A foreign key value must match an existing value in the parent table or be NULL.
- Foreign keys are based on data values and are purely logical, rather than physical, pointers.

# FOREIGN KEY Constraint

Define at either the table level or the column level:

```
CREATE TABLE employees (
    employee_id      NUMBER(6),
    last_name        VARCHAR2(25) NOT NULL,
    email            VARCHAR2(25),
    salary           NUMBER(8,2),
    commission_pct   NUMBER(2,2),
    hire_date        DATE NOT NULL,
    ...
    department_id    NUMBER(4),
    CONSTRAINT emp_dept_fk FOREIGN KEY (department_id)
        REFERENCES departments(department_id),
    CONSTRAINT emp_email_uk UNIQUE(email));
```

26

FOREIGN KEY constraints can be defined at the column or table constraint level. A composite foreign key must be created by using the table-level definition.

The example in the slide defines a FOREIGN KEY constraint on the DEPARTMENT\_ID column of the EMPLOYEES table, using table-level syntax. The name of the constraint is EMP\_DEPT\_FK.

The foreign key can also be defined at the column level, provided that the constraint is based on a single column. The syntax differs in that the keywords FOREIGN KEY do not appear, as shown in the following example:

```
CREATE TABLE employees
(
    ...
    department_id NUMBER(4) CONSTRAINT emp_deptid_fk
        REFERENCES departments(department_id),
    ...
)
```

# FOREIGN KEY Constraint: Keywords

- FOREIGN KEY: Defines the column in the child table at the table-constraint level
- REFERENCES: Identifies the table and column in the parent table
- ON DELETE CASCADE: Deletes the dependent rows in the child table when a row in the parent table is deleted
- ON DELETE SET NULL: Converts dependent foreign key values to null

27

O

For Instructor Use Only.  
This document should not be distributed.

The foreign key is defined in the child table and the column it references is in the parent table. The foreign key is defined using a combination of the following keywords:

- FOREIGN KEY is used to define the column in the child table at the table-constraint level.
- REFERENCES identifies the table and the column in the parent table.
- ON DELETE CASCADE indicates that when a row in the parent table is deleted, the dependent rows in the child table are also deleted.
- ON DELETE SET NULL indicates that when a row in the parent table is deleted, the foreign key values are set to null.

The default behavior is called the *restrict rule*, which disallows the update or deletion of referenced data.

Without the ON DELETE CASCADE or the ON DELETE SET NULL options, the row in the parent table cannot be deleted if it is referenced in the child table. Also, these keywords cannot be used in column-level syntax.

# CHECK Constraint

- Defines a condition that each row must satisfy
- Cannot reference columns from other tables

```
..., salary NUMBER(2)
CONSTRAINT emp_salary_min
    CHECK (salary > 0),...
```

28

O

For Instructor Use Only.  
This document should not be distributed.

When you define a `CHECK` constraint on a column, each row satisfies the condition. To satisfy the constraint, each row in the table must make the condition either TRUE or unknown (due to a null).

The condition can use the same constructs as the query conditions; however, they must not refer to other values in other rows.

A single column can have multiple `CHECK` constraints that refer to the column in its definition. There is no limit to the number of `CHECK` constraints that you can define on a column.

`CHECK` constraints can be defined at the column level or table level.

```
CREATE TABLE employees
(
    ...
    salary NUMBER(8,2) CONSTRAINT emp_salary_min
        CHECK (salary > 0),
    ...
)
```

## CREATE TABLE: Example

```
CREATE TABLE teach_emp (
    empno      NUMBER(5) PRIMARY KEY,
    ename      VARCHAR2(15) NOT NULL,
    job        VARCHAR2(10),
    mgr        NUMBER(5),
    hiredate   DATE DEFAULT (sysdate),
    photo      BLOB,
    sal        NUMBER(7,2),
    deptno    NUMBER(3) NOT NULL
        CONSTRAINT admin_dept_fkey
    REFERENCES
        departments(department_id));
```

29

0

The example in the slide shows the statement that is used to create the TEACH\_EMP table.

# Violating Constraints

Department 55 does not exist.

```
UPDATE employees  
SET department_id = 55  
WHERE department_id = 110;
```



```
Error starting at line : 1 in command -  
UPDATE employees  
SET department_id = 55  
WHERE department id = 110  
Error report -  
ORA-02291: integrity constraint (ORA1.EMP_DEPT_FK) violated - parent key not found
```

When you have constraints in place on columns, an error is returned if you try to violate the constraint rule. For example, if you try to update a record with a value that is tied to an integrity constraint, an error is returned.

In the example in the slide, department 55 does not exist in the parent table, DEPARTMENTS, and therefore, you receive the “parent key not found” violation ORA-02291.

# Violating Constraints

You cannot delete a row that contains a primary key that is used as a foreign key in another table.

```
DELETE FROM departments  
WHERE department_id = 60;
```



```
Error starting at line : 1 in command -  
delete from departments  
where department id = 60  
Error report -  
ORA-02292: integrity constraint (ORA1.EMP_DEPT_FK) violated - child record found
```

0

31

If you attempt to delete a record with a value that is tied to an integrity constraint, an error is returned.

The example in the slide tries to delete department 60 from the DEPARTMENTS table, but it results in an error because that department number is used as a foreign key in the EMPLOYEES table. If the parent record that you attempt to delete has child records, you receive the “child record found” violation ORA-02292.

The following statement works because there are no employees in department 70:

```
DELETE FROM departments  
WHERE department_id = 70;
```

```
1 row deleted.
```

# Lesson Agenda

- Database objects
  - Naming rules
- Data types
- CREATE TABLE statement
- Overview of constraints: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK constraints
- Creating a table using a subquery
- ALTER TABLE statement
- DROP TABLE statement

0



For Instructor Use Only.  
This document should not be distributed.

# Creating a Table Using a Subquery

- Create a table and insert rows by combining the `CREATE TABLE` statement and the `AS subquery` option.

```
CREATE TABLE table
  [(column, column...)]
AS subquery;
```

- Match the number of specified columns to the number of subquery columns.
- Define columns with column names and default values.



O

33

A second method for creating a table is to apply the `AS subquery` clause, which both creates the table and inserts rows returned from the subquery.

In the syntax:

<code>table</code>	Is the name of the table
<code>column</code>	Is the name of the column, default value, and integrity constraint
<code>subquery</code>	Is the <code>SELECT</code> statement that defines the set of rows to be inserted into the new table

## Guidelines

- The table is created with the specified column names, and the rows retrieved by the `SELECT` statement are inserted into the table.
- The column definition can contain only the column name and default value.
- If column specifications are given, the number of columns must equal the number of columns in the subquery `SELECT` list.
- If no column specifications are given, the column names of the table are the same as the column names in the subquery.
- The column data type definitions and the `NOT NULL` constraint are passed to the new table. Note that only the explicit `NOT NULL` constraint will be inherited. The `PRIMARY KEY` column will not pass the `NOT NULL` feature to the new column. Any other constraint rules are not passed to the new table. However, you can add constraints in the column definition.

# Creating a Table Using a Subquery

```
CREATE TABLE dept80
AS
SELECT employee_id, last_name,
       salary*12 ANNSAL,
       hire_date
  FROM employees
 WHERE department_id = 80;
```

Table DEPT80 created.

DESCRIBE dept80

Name	Null	Type
EMPLOYEE_ID		NUMBER(6)
LAST_NAME	NOT NULL	VARCHAR2(25)
ANNSAL		NUMBER
HIRE_DATE	NOT NULL	DATE

34

The example in the slide creates a table named DEPT80, which contains details of all the employees working in department 80. Notice that the data for the DEPT80 table comes from the EMPLOYEES table.

You can verify the existence of a database table and check the column definitions by using the DESCRIBE command.

However, be sure to provide a column alias when selecting an expression. The expression SALARY\*12 is given the alias ANNSAL. Without the alias, the following error is generated:

```
Error starting at line 1 in command:
CREATE TABLE dept80
AS
SELECT employee_id, last_name,
       salary*12,
       hire_date
  FROM employees
 WHERE department_id = 80
Error at Command Line:4 Column:18
Error report:
SQL Error: ORA-00998: must name this expression with a column alias
00998. 00000 -  "must name this expression with a column alias"
*Cause:
*Action:
```

# Lesson Agenda

- Database objects
  - Naming rules
- Data types
- CREATE TABLE statement
- Overview of constraints: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK constraints
- Creating a table using a subquery
- ALTER TABLE statement
- DROP TABLE statement

35

0



For Instructor Use Only.  
This document should not be distributed.

# ALTER TABLE Statement

Use the ALTER TABLE statement to:

- Add a new column
- Modify an existing column definition
- Define a default value for the new column
- Drop a column
- Rename a column
- Change table to read-only status

36

0

After you create a table, you may need to change the table structure for any of the following reasons:

- You omitted a column.
- Your column definition or its name needs to be changed.
- You need to remove columns.
- You want to put the table into read-only mode

You can do this by using the ALTER TABLE statement.

# ALTER TABLE Statement

Use the ALTER TABLE statement to add, modify, or drop columns:

```
ALTER TABLE table
ADD      (column datatype [DEFAULT expr]
[, column datatype]...);
```

```
ALTER TABLE table
MODIFY   (column datatype [DEFAULT expr]
[, column datatype]...);
```

```
ALTER TABLE table
DROP  (column [, column] ...);
```

You can add columns to a table, modify columns, and drop columns from a table by using the ALTER TABLE statement.

In the syntax:

<i>table</i>	Is the name of the table
ADD   MODIFY   DROP	Is the type of modification
<i>column</i>	Is the name of the column
<i>datatype</i>	Is the data type and length of the column
DEFAULT <i>expr</i>	Specifies the default value for a column

# Adding a Column

- You use the ADD clause to add columns:

```
ALTER TABLE dept80
ADD      (job_id VARCHAR2(9));
```

Table DEPT80 altered.

- The new column becomes the last column:

	EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE	JOB_ID
1	149	Zlotkey	126000	29-JAN-16	(null)
2	174	Abel	132000	11-MAY-12	(null)
3	176	Taylor	103200	24-MAR-14	(null)
4	206	Gietz	99600	07-JUN-10	(null)



0

38

## Guidelines for Adding a Column

- You can add or modify columns.
- You cannot specify where the column is to appear. The new column becomes the last column.

The example in the slide adds a column named `JOB_ID` to the `DEPT80` table. The `JOB_ID` column becomes the last column in the table.

**Note:** If a table already contains rows when a column is added, the new column is initially null or takes the default value for all the rows. You can add a mandatory `NOT NULL` column to a table that already contains data in the other columns only if you specify a default value. You can add a `NOT NULL` column to an empty table without the default value.

# Modifying a Column

- You can change a column's data type, size, and default value.

```
ALTER TABLE dept80
MODIFY
    (last_name VARCHAR2(30));
```

Table DEPT80 altered.

Size of the last\_name  
column is modified.

- A change to the default value of a column affects only subsequent insertions to the table.



O

39

You can modify a column definition by using the `ALTER TABLE` statement with the `MODIFY` clause. Column modification can include changes to a column's data type, size, and default value.

## Guidelines

- You can increase the width or precision of a numeric column.
- You can increase the width of character columns.
- You can decrease the width of a column if:
  - The column contains only null values
  - The table has no rows
  - The decrease in column width is not less than the existing values in that column
- You can change the data type if the column contains only null values. The exception to this is CHAR-to-VARCHAR2 conversions, which can be done with data in the columns.
- You can convert a CHAR column to the VARCHAR2 data type or convert a VARCHAR2 column to the CHAR data type only if the column contains null values or if you do not change the size.
- A change to the default value of a column affects only subsequent insertions to the table.

# Dropping a Column

Use the `DROP COLUMN` clause to drop columns that you no longer need from the table:

```
ALTER TABLE dept80
DROP (job_id);
```

Table DEPT80 altered.

	EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE
1	149	Twinkley	126000	29-JAN-16
2	174	Abel	132000	11-MAY-12
3	176	Taylor	103200	24-MAR-14
4	206	Gietz	99600	07-JUN-10



40

O

You can drop a column from a table by using the `ALTER TABLE` statement with the `DROP COLUMN` clause.

## Guidelines

- The column may or may not contain data.
- Using the `ALTER TABLE DROP COLUMN` statement, only one column can be dropped at a time.
- The table must have at least one column remaining in it after it is altered.
- After a column is dropped, it cannot be recovered.
- A primary key that is referenced by another column cannot be dropped, unless the cascade option is added.
- Dropping a column can take a while if the column has a large number of values. In this case, it may be better to set it to be unused and drop it when there are fewer users on the system to avoid extended locks.

**Note:** Certain columns can never be dropped, such as columns that form part of the partitioning key of a partitioned table or columns that form part of the `PRIMARY KEY` of an index-organized table. For more information about index-organized tables and partitioned tables, refer to the *Oracle Database Concepts* and *Oracle Database Administrator's Guide*.

# SET UNUSED Option

- You use the SET UNUSED option to mark one or more columns as unused.
- You use the DROP UNUSED COLUMNS option to remove the columns that are marked as unused.
- You can specify the ONLINE keyword to indicate that DML operations on the table will be allowed while marking the column or columns UNUSED.

```
ALTER TABLE <table_name>
SET UNUSED(<column_name> [, <column_name>]);
OR
ALTER TABLE <table_name>
SET UNUSED COLUMN <column_name> [, <column_name>];
```

```
ALTER TABLE <table_name>
DROP UNUSED COLUMNS;
```

41

You can use the SET UNUSED option to mark one or more columns as unused so that they can be dropped when the demand on system resources is lower. Specifying this clause does not actually remove the target columns from each row in the table (that is, it does not restore the disk space used by these columns). Therefore, the response time is faster than if you executed the DROP clause.

Unused columns are treated as if they were dropped, even though their column data remains in the table's rows. After a column has been marked as unused, a SELECT \* query will not retrieve data from unused columns. In addition, the names and types of columns marked as unused will not be displayed during a DESCRIBE statement, and you can add to the table a new column with the same name as an unused column. The SET UNUSED information is stored in the USER\_UNUSED\_COL\_TABS dictionary view.

You can specify the ONLINE keyword to indicate that DML operations on the table will be allowed while marking the column or columns UNUSED. The code example shows the use of SET UNUSED COLUMN that sets a column unused forever using the ONLINE keyword.

```
ALTER TABLE dept80 SET UNUSED(hire_date) ONLINE;
```

**Note:** The guidelines for setting a column to be UNUSED are similar to those for dropping a column.

### **DROP UNUSED COLUMNS Option**

DROP UNUSED COLUMNS removes from the table all columns that are currently marked as unused. You can use this statement when you want to reclaim the extra disk space from the unused columns in the table. If the table contains no unused columns, the statement returns with no errors.

```
ALTER TABLE dept80  
SET UNUSED (last_name);
```

```
ALTER TABLE dept80  
DROP UNUSED COLUMNS;
```

**Note:** A subsequent DROP UNUSED COLUMNS will physically remove all unused columns from a table, similar to a DROP COLUMN.

# Read-Only Tables

You can use the ALTER TABLE syntax to:

- Put a table in read-only mode, which prevents DDL or DML changes during table maintenance
- Put the table back into read/write mode

```
ALTER TABLE employees READ ONLY;  
  
-- perform table maintenance and then  
-- return table back to read/write mode  
  
ALTER TABLE employees READ WRITE;
```



43

You can specify READ ONLY to convert a table into read-only mode. When the table is in READ ONLY mode, you cannot issue any DML statements that affect the table or any SELECT . . . FOR UPDATE statements. You can issue DDL statements as long as they do not modify any data in the table. Operations on indexes associated with the table are allowed when the table is in READ ONLY mode.

Specify READ/WRITE to return a read-only table to read/write mode.

**Note:** You can drop a table that is in READ ONLY mode. The DROP command is executed only in the data dictionary, so access to the table contents is not required. The space used by the table will not be reclaimed until the tablespace is made read/write again, and then the required changes can be made to the block segment headers, and so on.

# Lesson Agenda

- Database objects
  - Naming rules
- Data types
- CREATE TABLE statement
- Overview of constraints: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK constraints
- Creating a table using a subquery
- ALTER TABLE statement
- DROP TABLE statement

44

0



For Instructor Use Only.  
This document should not be distributed.

# Dropping a Table

- Moves a table to the recycle bin
- Removes the table and all its data entirely if the PURGE clause is specified
- Invalidates dependent objects and removes object privileges on the table

```
DROP TABLE dept80;  
Table DEPT80 dropped.
```



O

45

The `DROP TABLE` statement moves a table to the recycle bin or removes the table and all its data from the database entirely. Unless you specify the `PURGE` clause, the `DROP TABLE` statement does not result in space being released back to the tablespace for use by other objects, and the space continues to count toward the user's space quota. Dropping a table invalidates the dependent objects and removes object privileges on the table.

When you drop a table, the database loses all the data in the table and all the indexes associated with it.

## Syntax

```
DROP TABLE table [PURGE]
```

In the syntax, *table* is the name of the table.

## Guidelines

- All data is deleted from the table.
- Any views and synonyms remain, but are invalid.
- Any pending transactions are committed.
- Only the creator of the table or a user with the `DROP ANY TABLE` privilege can remove a table.

**Note:** Use the `FLASHBACK TABLE` statement to restore a dropped table from the recycle bin. This is discussed in detail in the course titled *Oracle Database 19c: SQL Workshop II*.

# Summary

In this lesson, you should have learned how to use the CREATE TABLE, ALTER TABLE, and DROP TABLE statements to create a table, modify a table and columns, and include constraints.

- Categorize the main database objects
- Review the table structure
- List the data types that are available for columns
- Create a simple table
- Explain how constraints are created at the time of table creation



46

In this lesson, you should have learned the following:

## **CREATE TABLE**

- Use the CREATE TABLE statement to create a table and include constraints.
- Create a table based on another table by using a subquery.

## **DROP TABLE**

- Remove rows and a table structure.
- When executed, this statement cannot be rolled back.

# Practice 11a: Overview

This practice covers the following topics:

- Creating new tables
- Creating a new table by using the `CREATE TABLE AS` syntax
- Verifying that tables exist
- Altering tables
- Adding columns
- Dropping columns
- Setting a table to read-only status
- Dropping tables



O

47

You create new tables by using the `CREATE TABLE` statement and confirm that the new table was added to the database. You also learn to set the status of a table as `READ ONLY`, and then revert to `READ/WRITE`.

**Note:** For all DDL and DML statements, click the Run Script icon (or press F5) to execute the query in SQL Developer. Thus, you get to see the feedback messages on the Script Output tabbed page. For `SELECT` queries, continue to click the Execute Statement icon or press F9 to get the formatted output on the Results tabbed page.

For Instructor Use Only.  
This document should not be distributed.

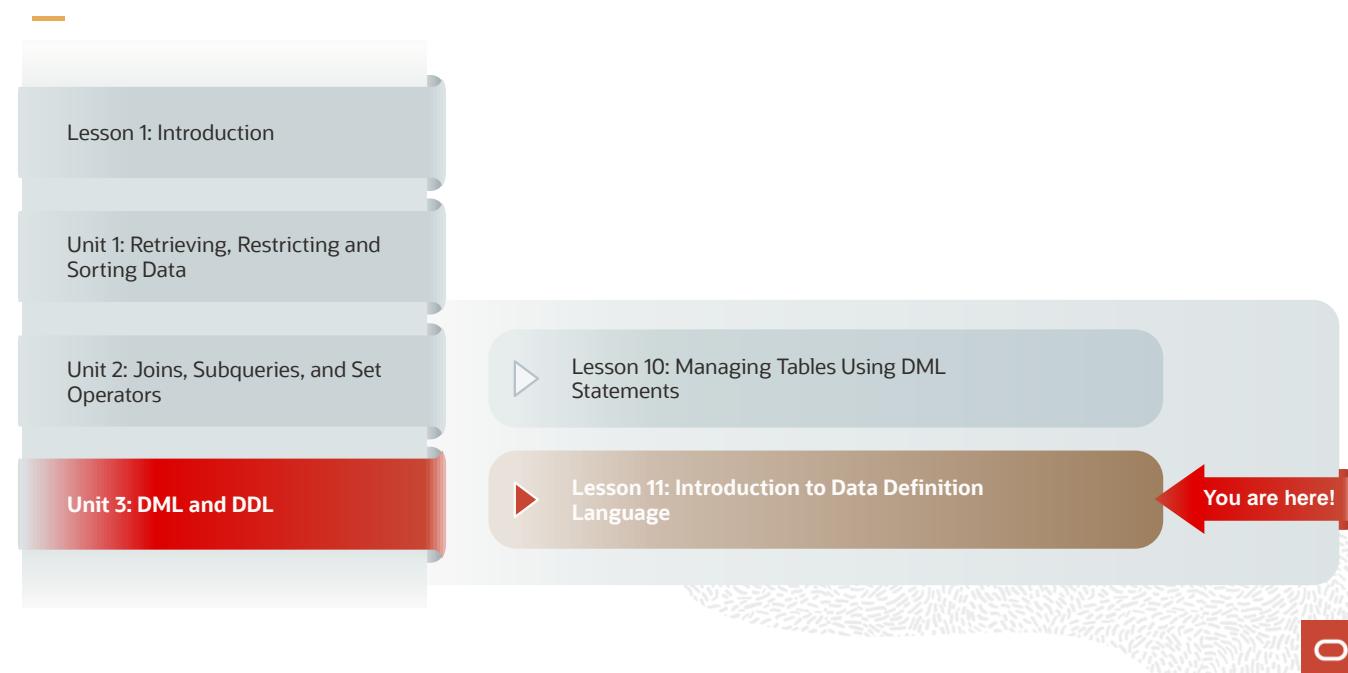
Using the Regular Expressions Functions and Conditions in SQL and PL/SQL	I-5
What are Metacharacters?	I-6
Using Metacharacters with Regular Expressions	I-7
Regular Expressions Functions and Conditions: Syntax	I-9
Performing a Basic Search by Using the REGEXP_LIKE Condition	I-10
Replacing Patterns by 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

## Introduction to Data Definition Language in MySQL

0

For Instructor Use Only.  
This document should not be distributed.

# Course Roadmap



2

In Unit 3, you learn how to create and manage database objects using data definition language (DDL) statements. You will also learn how to manage data in the tables using data manipulation language (DML) statements.

# Objectives

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

- Create a database
- Add tables to a database
- List the data types that are available for columns
- Use the SHOW CREATE TABLE statement
- Describe and set column and table options
- Create indexes, keys, and constraints

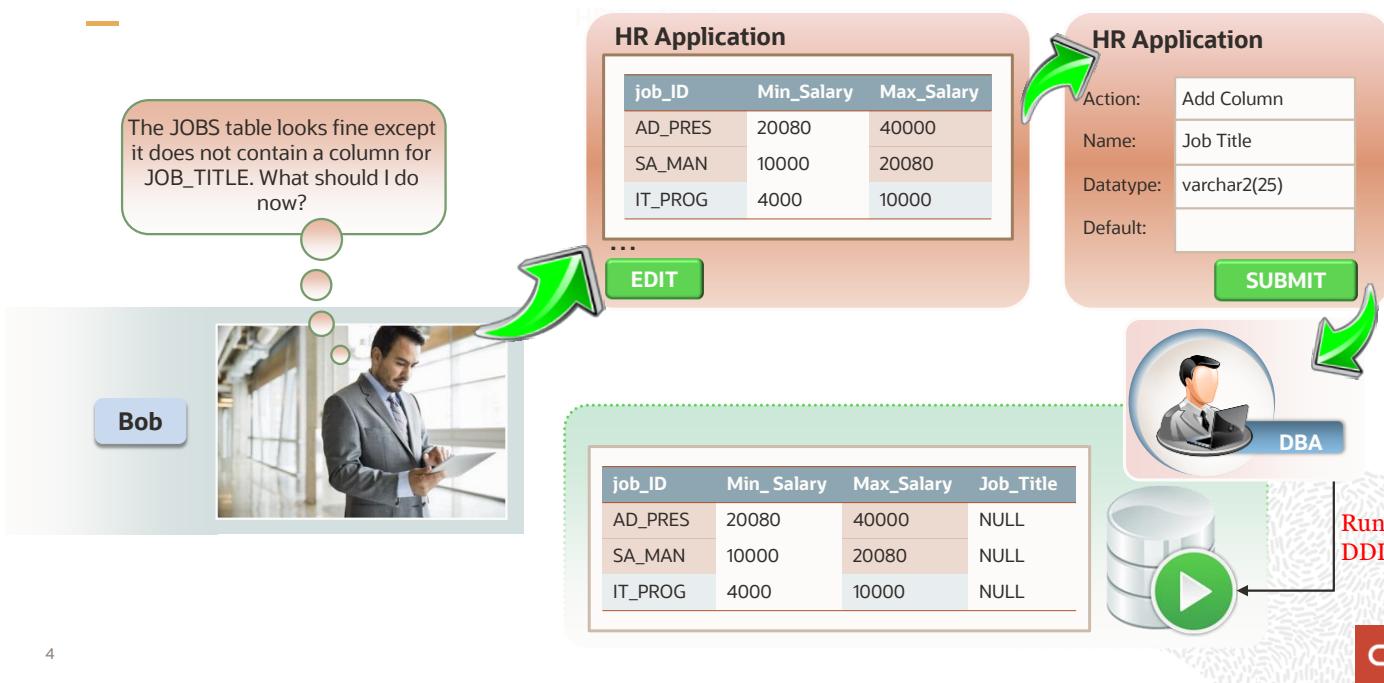


0

3

In this lesson, you are introduced to the data definition language (DDL) statements. You learn the basics of creating simple tables, altering them, and removing them. The data types available in DDL are shown and schema concepts are introduced. Constraints are discussed in this lesson. Exception messages that are generated from violating constraints during DML operations are shown and explained.

# HR Application Scenario



Consider a scenario where Bob, an HR manager is creating tables to store all the employee information in the database. While creating the JOBS table to store all the job information, he forgets to create a column for the job title. So what should Bob do now?

Should he drop the table and create JOBS table again? No!

Bob can alter the table and add a new column (JOB\_TITLE) to the existing JOBS table. The statements that allow you to modify the structure of database objects are called data definition language (DDL) statements. Usually, the permission to execute DDL statements is given only to the admin.

Bob submits the request to alter the JOBS table structure along with the details. The DBA receives the request and constructs an appropriate SQL statement to alter the JOBS table.

A new column called Job\_Title is added to the JOBS table. The value for Job\_Title remains NULL until Bob goes to the application and updates the values for all the records.

In this lesson, you learn more about DDL statements.

# Lesson Agenda

- Database objects
  - Naming rules
- **CREATE TABLE statement**
- Data types
- Indexes, keys, and constraints
- Column options
- Creating a table using a subquery
- ALTER TABLE statement
- DROP TABLE statement

5

0



For Instructor Use Only.  
This document should not be distributed.

# Creating a Database: Syntax

General syntax for creating a database:

```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

Or:

```
CREATE SCHEMA [IF NOT EXISTS] database_name;
```

The IF NOT EXISTS option prevents getting an error if the database already exists.

# MySQL Naming Conventions

Database names, table names, and column names:

- Can contain A–Z, a–z, 0–9, \_, and \$
- Cannot have more than 64 characters
- Cannot contain reserved words or special characters, such as /, \, ., # or spaces unless you enclose the name with backticks

For example, the following causes an error:

```
CREATE DATABASE my database;
```

Correct the error by enclosing the name in backticks:

```
CREATE DATABASE `my database`;
```

Case sensitivity of database and table names depends on the host operating system.

7

0

Databases and tables in MySQL are created by using directories and files in the host file system. Because of this, if the host operating system is case-sensitive in its treatment of directory and file names, so is MySQL. Windows file names are not case-sensitive, so a server running on Windows does not treat database and table names as case-sensitive. MySQL servers running on UNIX usually treat database and table names as case-sensitive because UNIX file names are case-sensitive.

Table names can be reserved words or contain special characters if the name is quoted in backticks ( `` ).

# Lesson Agenda

- Database objects
  - Naming rules
- **CREATE TABLE statement**
- Data types
- Indexes, keys, and constraints
- Column options
- Creating a table using a subquery
- ALTER TABLE statement
- DROP TABLE statement

0



# CREATE TABLE Statement

General syntax for creating a table:

```
CREATE TABLE [database.]table  
  (column_name datatype [, ...]);
```

- Example:

```
CREATE TABLE employees1  
  (employee_id      INTEGER,  
   first_name       VARCHAR(20),  
   last_name        VARCHAR(25),  
   email            VARCHAR(25),  
   phone_number     VARCHAR(20),  
   hire_date        DATE,  
   job_id           VARCHAR(10),  
   salary            DECIMAL(8,2),  
   commission_pct   DECIMAL(2,2),  
   manager_id       INTEGER,  
   department_id    INTEGER);
```



0

9

You create tables to store data by executing the SQL CREATE TABLE statement. This statement is one of the DDL statements that are used to create, modify, or remove database structures. These statements record information in the data dictionary. The data dictionary is an important set of read-only tables that provide database information.

To create a table, a user must have the CREATE TABLE privilege. The example in this slide creates a table similar to the employees table. It creates a table with columns and their datatypes specified. This lesson describes numerous other options that can be used when creating tables. The table in this slide uses the data types in the employee table. MySQL supports many other data types.



# Lesson Agenda

- Database objects
  - Naming rules
- **CREATE TABLE statement**
- Data types
- Indexes, keys, and constraints
- Column options
- Creating a table using a subquery
- ALTER TABLE statement
- DROP TABLE statement



0



10

For Instructor Use Only.  
This document should not be distributed.

# Data Types: Overview

- Numeric data types
  - Integer, decimal, and floating point data
- Date and time data types
  - Calendar dates, with or without time data and time intervals
- String data types
  - Character strings and BLOBs
- Spatial data types
  - Data representing geographic features
- JavaScript Object Notation (JSON)
  - Structured text data stored in JSON documents

# Numeric Data Types

Integer

- Positive or negative whole numbers
  - Syntax: INT [ (M) ] [UNSIGNED] [ZEROFILL]

Fixed Point

- Used for exact-value numbers: integer, fractional, or both
  - Syntax: DECIMAL [ (M[, D]) ] [UNSIGNED] [ZEROFILL]

## Floating Point

- Used for approximate-value numbers: integer, fractional, or both
  - Syntax: FLOAT [ (M, D) ] [UNSIGNED] [ZEROFILL]

12

6

In the syntax,  $M$  represents an optional value for the maximum number of digits available, and, for DECIMAL and FLOAT,  $D$  represents the number of digits following the decimal point. INTEGER is a synonym for INT. For INT, if  $M$  is omitted, a default number of digits is provided.

`NUMERIC` and `DEC` are synonyms for `DECIMAL`. For `DECIMAL` data type, if `D` is omitted, the default is 0 decimal places, and if `M` is omitted, the default is 10 significant digits. The decimal point and (for negative numbers) the minus (-) sign are not counted in `M`.

Floating point numbers are used to represent very large or very small numbers (close to zero) using a small amount of space. For example  $6.022169 \times 10^{23}$  represents  $602216900000000000000000$ , while  $6.022169 \times 10^{-23}$  represents  $0.0000000000000006022169$  as a value. If  $M$  and  $D$  are omitted, FLOAT values are stored to the limits permitted by the hardware.

MySQL supports a number of other integer, fixed point, and floating point data types for efficiency of storage or calculations. Other data types are covered in the course on MySQL Essentials.

# Date and Time Data Types

## DATE

- YYYY-MM-DD as in 2018-01-04

## TIME

- HH:MM:SS as in 12:59:02

## DATETIME

- YYYY-MM-DD HH:MM:SS as in 2018-01-04 12:59:02

13

Here, YYYY, MM, DD, HH, MM, and SS stand for year, month, day of month, hour, minute, and second, respectively.

You have used the DATE data type previously in this course.

The TIME data type can express time of day or elapsed time.

The DATETIME data type is a combination of date and time.

The TIME and DATETIME data types can include fractional seconds up to 6 decimal places.

Other date and time data types and the use of fractional seconds are covered in the course on MySQL Essentials.

# String Data Types

- Fixed length character strings: CHAR ( $M$ )
  - Stored as fixed length  $M$ , right-padded with spaces
- Variable length character strings: VARCHAR ( $M$ )
  - Stored **up to** length  $M$ , **not** right-padded with spaces
- Extremely long character strings: TEXT
  - Variable length character strings, stored as a separately allocated object
- Binary large objects: BLOB
  - Variable length binary data, stored as a separately allocated object

# Lesson Agenda

- Database objects
  - Naming rules
- **CREATE TABLE statement**
- Data types
- Indexes, keys, and constraints
- Column options
- Creating a table using a subquery
- ALTER TABLE statement
- DROP TABLE statement

15

0



For Instructor Use Only.  
This document should not be distributed.

# Indexes, Keys, and Constraints

- Index: A column or columns of table data designated to be used as pointers to table rows to speed up data retrieval
- Key: Normally a synonym for index in MySQL
- Constraint: A restriction placed on one or more column values of a table to enforce integrity rules

The following MySQL indexes (keys) include designated combinations of constraints:

- Primary Keys
- Unique Keys
- Foreign Keys

A constraint is a condition that data must meet when being added to or removed from a table. For example, a unique constraint requires all values in an index to be distinct. If you try to add a new row with a value in the index that matches a value in an existing row, an error occurs. Primary keys, unique keys, and foreign keys are indexes that have specific combinations of constraints. The sections in this lesson titled "Primary Keys", "Unique Key Constraints", and "Foreign Key Constraints" describe the constraints specific to these indexes and how you can include these types of indexes in a table.

# Table Indexes

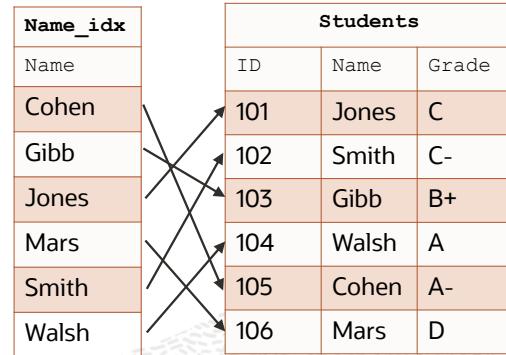
An index is a collection of pointers to records in a table.

An index helps to:

- Locate rows quickly
- Avoid full table scans
- Improve query performance

Unnecessary indexes are wasteful.

- They take up space in the database.
- They are updated whenever data is updated.



17

By default, when MySQL attempts to locate a record, it scans the entire table until a match is found (full table scan). This can really slow down large queries.

An index helps MySQL find specific column values quickly. The index entries act as pointers to the rows that match these criteria so they can be accessed without a full table scan.

You can assign indexes when you create a table, or at any time afterwards. You can create indexes on single columns or multiple columns (composite indexes). For example, you can use the phone number for an individual in a phone book as a single column index, or you can use the last name and first name to create a composite index.

You can index any table column, but do not overdo it. Indexes take up space. Too many indexes can slow performance because indexes are updated whenever you add or delete rows or modify data in indexed columns. Avoid indexing fields that change regularly. Selecting columns for indexing is covered in the MySQL Database Administrators course.

**Note:** A B-tree is a type of data structure used by DBMSs to store indexes. The data is kept sorted at all times, which makes for fast lookup of exact matches or comparisons of values.

# Primary Keys

A primary key has the following constraints:

- Uniquely identifies a single row in the table
- Null values not allowed
- Only one primary key allowed per table

```
CREATE TABLE employees2
  (employee_id      INTEGER,
   first_name       VARCHAR(20),
   last_name        VARCHAR(25),
   email            VARCHAR(25),
   phone_number     VARCHAR(20),
   hire_date        DATE,
   job_id           VARCHAR(10),
   salary            DECIMAL(8,2),
   commission_pct   DECIMAL(2,2),
   manager_id       INTEGER,
   department_id    INTEGER,
   PRIMARY KEY (employee_id);
```

The primary key is the identifying key for the table. Each row in the table must have a unique value for the primary key and it cannot be NULL. The primary key can be a composite key, which is covered elsewhere in this lesson. The InnoDB engine stores tables sorted by the primary key.

The example in this slide uses the `PRIMARY KEY` clause at the end of the statement. Another option is to use the `PRIMARY KEY` attribute in the column definition for `ID`, as shown in this example:

```
CREATE TABLE employees2
  (employee_id      INTEGER PRIMARY KEY,
   first_name       VARCHAR(20),
   last_name        VARCHAR(25),
   email            VARCHAR(25),
   phone_number     VARCHAR(20),
   hire_date        DATE,
   job_id           VARCHAR(10),
   salary            DECIMAL(8,2),
   commission_pct   DECIMAL(2,2),
   manager_id       INTEGER,
   department_id    INTEGER);
```

# Unique Key Constraints

Unique keys have the following constraints:

- The value must be unique for that column within the table.
- Null values are allowed for unique keys.
- Multiple unique keys are allowed per table.

```
CREATE TABLE employees3
  (employee_id      INTEGER,
   first_name       VARCHAR(20),
   last_name        VARCHAR(25),
   email            VARCHAR(25),
   phone_number     VARCHAR(20),
   hire_date        DATE,
   job_id           VARCHAR(10),
   salary           DECIMAL(8,2),
   commission_pct   DECIMAL(2,2),
   manager_id       INTEGER,
   department_id    INTEGER,
   PRIMARY KEY (employee_id),
   UNIQUE KEY (email));
```

19

The unique key constraint requires that all values in the column are different. Primary keys are unique keys that have the additional constraint that null values are not allowed and only one primary key is allowed per table. You can specify unique keys that do not have the additional constraints of primary keys. In the example in this slide, each employee's email must be unique. The example in this slide uses `UNIQUE KEY` clauses at the end of the statement. Another option is to use `UNIQUE KEY` attributes in the column definitions, as shown in this example:

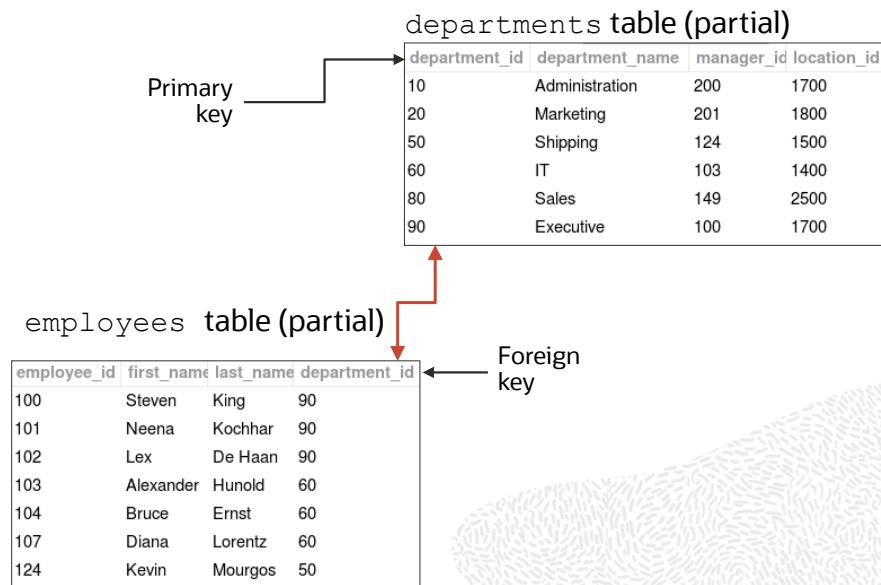
```
CREATE TABLE employees3
  (employee_id      INTEGER PRIMARY KEY,
   first_name       VARCHAR(20),
   last_name        VARCHAR(25),
   email            VARCHAR(25) UNIQUE KEY,
   phone_number     VARCHAR(20),
   hire_date        DATE,
   job_id           VARCHAR(10),
   salary           DECIMAL(8,2),
   commission_pct   DECIMAL(2,2),
   manager_id       INTEGER,
   department_id    INTEGER);
```

# Foreign Key Constraints

Foreign keys maintain referential integrity by enforcing a link between the data in two tables:

- The foreign key in the child table references the primary key in the parent table.
- The foreign key constraint prevents actions that would destroy the links between the child and parent tables. For example:
  - When inserting or updating a row in a child table, the value for the foreign key column being inserted or updated must already exist as the primary key for a row in the parent table. Otherwise the insert or update is rejected.
  - When deleting or updating a row in the parent table, the foreign key constraint determines what happens to any rows in the child table with a foreign key matching the column being deleted or updated.

# Foreign Key Constraint: Example Tables



21

0

The foreign key (or referential integrity) constraint designates a column or a combination of columns as a foreign key, and establishes a relationship with a primary key or a unique key in the same table or a different table.

In the example in the slide, `department_id` has been defined as the foreign key in the `employees` table (child table); it references the `department_id` column of the `departments` table (the parent table). If you try to insert a row into the `employees` table or change the `department_id` of a row in the `employees` table, the value provided for `department_id` must already be a primary key in the `departments` table. If you try to delete a row from the `departments` table or try to change the value for a `department_id` in the `departments` table it will first check whether there is a row in the `employees` table with that foreign key and take the action defined for the foreign key, described later in this lesson.

## Guidelines

- A foreign key value must match an existing value in the parent table or be `NULL`.
- Foreign keys are based on data values and are purely logical, rather than physical, pointers.
- The primary key and foreign key might have the same column name (as in this example), but that is not required and often not appropriate.

# FOREIGN KEY Constraint: Example Statement

```
CREATE TABLE employees4
  (employee_id      INTEGER,
   first_name       VARCHAR(20),
   last_name        VARCHAR(25),
   email            VARCHAR(25),
   phone_number     VARCHAR(20),
   hire_date        DATE,
   job_id           VARCHAR(10),
   salary            DECIMAL(8,2),
   commission_pct   DECIMAL(2,2),
   manager_id       INTEGER,
   department_id    INTEGER,
   PRIMARY KEY (employee_id),
   UNIQUE KEY (email),
   CONSTRAINT emp4_dept_fk
     FOREIGN KEY (department_id)
     REFERENCES departments (department_id));
```

22

0

The example in the slide defines a FOREIGN KEY constraint on the department\_id column of the employees table. This makes department\_id in the employees4 child table a foreign key that must match a value for the primary key department\_id in the departments table. The name of the constraint is emp4\_dept\_fk. The name must be unique for all tables in the database. If you do not provide a name for the foreign key constraint, MySQL will generate a name. The CONSTRAINT keyword is optional. If CONSTRAINT is omitted, the foreign key name must be omitted.

# FOREIGN KEY Constraint: Referential Actions

To control what happens to rows in a child table when a `DELETE` or `UPDATE` operation affects a key value in a parent table that has matching rows in the child table, you can include one or both of these optional subclauses at the end of the `FOREIGN KEY` clause:

```
[ON DELETE referential_action]  
[ON UPDATE referential_action]
```

The `referential_action` can be one of the following:

- `RESTRICT` or `NO ACTION`: This rejects the `DELETE` or `UPDATE` operation on the parent table. It is the default action if `ON DELETE` or `ON UPDATE` is omitted.
- `CASCADE`: This deletes or updates the row in the parent table and automatically deletes or updates the matching rows in the child table.
- `SET NULL`: This deletes or updates the row in the parent table and sets the foreign key column or columns in the child table to `NULL`.

If `ON DELETE` or `ON UPDATE` are not included at the end of a `FOREIGN KEY` clause, the default is to reject a deletion or update on the parent table. This is the same as specifying `RESTRICT` or `NO ACTION`. Specifying `ON DELETE CASCADE` in the example with the `department_id` foreign key would mean that, if you delete a department from the `departments` table, all employees with that same `department_id` would be deleted. Specifying `ON UPDATE CASCADE` in the example would mean that, if you change the `department_id` in the `department` table, all employees with that `department_id` would be assigned the newly updated `department_id`. If you specify `ON DELETE SET NULL` or `ON UPDATE SET NULL`, be sure the column in the child table is not defined as `NOT NULL`.

# Secondary Indexes

A secondary index does not have any constraints and is used to speed up access to data. For example:

```
CREATE TABLE employees5
  (
    employee_id      INTEGER,
    first_name       VARCHAR(20),
    last_name        VARCHAR(25),
    email            VARCHAR(25),
    phone_number     VARCHAR(20),
    hire_date        DATE,
    job_id           VARCHAR(10),
    salary            DECIMAL(8,2),
    commission_pct   DECIMAL(2,2),
    manager_id       INTEGER,
    department_id    INTEGER,
    PRIMARY KEY (employee_id),
    UNIQUE KEY (email),
    INDEX emp5_name_ix (last_name),
    CONSTRAINT emp5_dept_fk
      FOREIGN KEY (department_id)
      REFERENCES departments (department_id);
```

A secondary index (or key) does not have to be unique or have any referential constraints. The index provides faster access to data. For example, if you know that many searches will be done on employees' last names, you can create an index for the `last_name` column. Last names do not have to be unique. Any kind of index, including primary key, unique key, foreign key, or secondary index can be a composite key combining multiple columns to form the key value. For primary and unique keys, the individual parts of the composite key do not have to be unique, but the value of the combined columns must be unique.

# Lesson Agenda

- Database objects
  - Naming rules
- **CREATE TABLE statement**
- Data types
- Indexes, keys, and constraints
- Column options
- Creating a table using a subquery
- ALTER TABLE statement
- DROP TABLE statement

25

0



For Instructor Use Only.  
This document should not be distributed.

# Column Options

You can add options to the `CREATE TABLE` statement's column definitions, including `NULL`, `NOT NULL`, `DEFAULT`, or `AUTO_INCREMENT`.

Example:

```
CREATE TABLE employees6
  (employee_id      INTEGER AUTO_INCREMENT,
   first_name       VARCHAR(20),
   last_name        VARCHAR(25) NOT NULL,
   email            VARCHAR(25) NOT NULL,
   phone_number     VARCHAR(20),
   hire_date        DATE NOT NULL,
   job_id           VARCHAR(10) NOT NULL,
   salary            DECIMAL(8,2),
   commission_pct   DECIMAL(2,2),
   manager_id       INTEGER DEFAULT 1,
   department_id    INTEGER DEFAULT 10,
   PRIMARY KEY (employee_id),
   UNIQUE KEY (email),
   INDEX emp6_name_ix (last_name),
   CONSTRAINT emp6_dept_fk
     FOREIGN KEY (department_id)
     REFERENCES departments (department_id));
```

26

0

The example in this slide creates a table similar to the `employees` table. You can include `NULL` as an option or omit it because it is the default and means that the column can accept `NULL` as a value. Specifying `NOT NULL` indicates that the column must have a value and cannot be set to `NULL`. The `DEFAULT` option specifies the default value to use when a row is inserted. To use the default value, if specifying the list of column names in the `INSERT` statement, omit the column name for the column to be set to its default and don't provide a value for it in the `VALUES` clause. Alternatively, you can use the `DEFAULT` keyword in the `VALUES` clause to set the column to its default value.

The `AUTO_INCREMENT` attribute can be set on an indexed column and sets the value for the column to the next higher value. When inserting a new value, either omit the value from the list of column names and `VALUES` clause or specify the value as `NULL`, and it will be assigned the next higher value.

The following statement would set the `employee_id` to the next higher integer value, set `manager_id` and `department_id` to their default values of 1 and 10, respectively, and set any columns not listed to `NULL`.

```
INSERT INTO employees6 (last_name, email, hire_date, job_id)
  VALUES ('jones', 'jones', '2010-01-01', 'AC_MGR');
```

# Lesson Agenda

- Database objects
  - Naming rules
- **CREATE TABLE statement**
- Data types
- Indexes, keys, and constraints
- Column options
- Creating a table using a subquery
- ALTER TABLE statement
- DROP TABLE statement

27

0



For Instructor Use Only.  
This document should not be distributed.

# Creating a Table Using a Subquery

- Create a table and insert rows by combining the `CREATE TABLE` statement and the `AS subquery` option.

```
CREATE TABLE table
  [(column, column...)]
AS subquery;
```

- Match the number of specified columns to the number of subquery columns.
- Define columns with column names and default values.



28

A second method for creating a table is to apply the `AS subquery` clause, which both creates the table and inserts rows returned from the subquery.

In the syntax:

- `table`: the name of the table
- `column`: the name of the column, default value, and integrity constraint
- `subquery`: the `SELECT` statement that defines the set of rows to be inserted into the new table

## Guidelines

- The table is created with the specified column names, and the rows retrieved by the `SELECT` statement are inserted into the table.
- The column definition can contain only the column name and default value.
- If column specifications are given, the number of columns must equal the number of columns in the subquery `SELECT` list.
- If no column specifications are given, the column names of the table are the same as the column names in the subquery.
- The column data type definitions and the `NOT NULL` constraint are passed to the new table. Note that only the explicit `NOT NULL` constraint will be inherited. The `PRIMARY KEY` column will not pass the `NOT NULL` feature to the new column. Any other constraint rules are not passed to the new table. However, you can add constraints in the column definition.

# Creating a Table Using a Subquery: Example

```
CREATE TABLE    dept80
AS
SELECT  employee_id, last_name,
        salary*12 ANNSAL,
        hire_date
FROM    employees
WHERE   department_id = 80;
```

```
DESCRIBE dept80;
```

#	Field	Type	Null	Key	Default	Extra
1	employee_id	int(11)	NO		NULL	
2	last_name	varchar(25)	NO		NULL	
3	ANNSAL	decimal(10,2)	YES		NULL	
4	hire_date	date	NO		NULL	

```
SELECT * FROM dept80;
```

#	employee_id	last_name	ANNSAL	hire_date
1	149	Zlotkey	126000.00	2016-01-29
2	174	Abel	132000.00	2012-05-11
3	176	Taylor	103200.00	2014-03-24

29

0

The example in the slide creates a table named `dept80`, which contains details of all the employees working in department 80. Notice that the data for the `dept80` table comes from the `employees` table. You can verify the existence of a database table and check the column definitions by using the `DESCRIBE` statement. The `employee_id` column is not set as the primary key, but the `NOT NULL` attributes are maintained. You can provide a column alias when using an expression in the subquery. In the example, the expression `salary*12` is given the alias `ANNSAL`.

# Lesson Agenda

- Database objects
  - Naming rules
- **CREATE TABLE statement**
- Data types
- Indexes, keys, and constraints
- Column options
- Creating a table using a subquery
- **ALTER TABLE statement**
- **DROP TABLE statement**

30

0



For Instructor Use Only.  
This document should not be distributed.

# ALTER TABLE Statement

Use the ALTER TABLE statement to:

- Add a new column
- Modify an existing column definition
- Define a default value for the new column
- Drop a column
- Rename a column
- Add an index or constraint

31

0

After you create a table, you may need to change the table structure for any of the following reasons:

- You omitted a column.
- Your column definition or its name needs to be changed.
- You need to remove columns.
- You want to put the table into read-only mode

You can do this by using the ALTER TABLE statement.

## ALTER TABLE Statement: Add, Modify, or Drop Columns

Use the ALTER TABLE statement to add, modify, or drop columns:

```
ALTER TABLE table
ADD [COLUMN] column datatype [column_options]
[FIRST | AFTER column_name];
```

```
ALTER TABLE table
MODIFY [COLUMN] column datatype [column_options]
[FIRST | AFTER column_name];
```

```
ALTER TABLE table
DROP [COLUMN] column;
```

You can add columns to a table, modify columns, and drop columns from a table by using the ALTER TABLE statement.

# Adding a Column

- You use the ADD clause to add columns:

```
ALTER TABLE dept80
ADD job_id VARCHAR(10) NOT NULL DEFAULT 'ST_CLERK'
AFTER last_name;
```

- The new job\_id column is added after the last\_name column, with the values set to ST\_CLERK:

```
SELECT * FROM dept80;
```

#	employee_id	last_name	job_id	ANNSAL	hire_date
1	149	Zlotkey	ST_CLERK	126000.00	2016-01-29
2	174	Abel	ST_CLERK	132000.00	2012-05-11
3	176	Taylor	ST_CLERK	103200.00	2014-03-24



O

33

The example in the slide adds a column named job\_id to the dept80 table.. Because this table already contains data, in order to set the NOT NULL option, you need to provide a default value, and all the rows are provided with that default value in that column. If the table is empty, you can specify the NOT NULL option without providing a default value. If you do not specify the NOT NULL or DEFAULT options, the values of the new column are set to NULL. In the example, the job\_id column is inserted after the last\_name column. If you specify FIRST rather than AFTER, the column is added before the employee\_id column. If you do not specify either FIRST or AFTER, by default, the column is added at the end.

# Modifying a Column

- You can change a column's data type, size, column options or position.

```
ALTER TABLE dept80
MODIFY last_name VARCHAR(30) NOT NULL FIRST;
```

```
DESCRIBE dept80;
```

#	Field	Type	Null	Key	Default	Extra
1	last_name	varchar(30)	NO		NULL	
2	employee_id	int(11)	NO		NULL	
3	job_id	varchar(10)	NO		ST_CLERK	
4	ANNSAL	decimal(10,2)	YES		NULL	
5	hire_date	date	NO		NULL	

- The size of the `last_name` column is modified and it is moved to the first column.
- The full column definition must be included. If `NOT NULL` had been omitted, the column would have been changed from `NOT NULL` to accepting `NULL` values.
- If a default value is provided, it affects only subsequent insertions to the table

You can modify a column definition by using the `ALTER TABLE` statement with the `MODIFY` clause. Column modification can include changes to a column's data type, size, or column options like `DEFAULT` and `NOT NULL`. You can also change the position of the column with `FIRST` or `AFTER`. The column remains in its previous position if you omit `FIRST` or `AFTER`.

# Dropping a Column

Use the `DROP COLUMN` clause to drop columns that you no longer need from the table:

```
ALTER TABLE dept80
DROP COLUMN job_id;
```

```
SELECT * FROM dept80;
```

#	last_name	employee_id	ANNSAL	hire_date
1	Zlotkey	149	126000.00	2016-01-29
2	Abel	174	132000.00	2012-05-11
3	Taylor	176	103200.00	2014-03-24



O

35

You can drop a column from a table by using the `ALTER TABLE` statement with the `DROP COLUMN` clause. If a table includes only one column, the column cannot be dropped. A MySQL extension to standard SQL permits a single `ALTER TABLE` statement to contain multiple `ADD`, `MODIFY`, or `DROP` clauses, separated by commas. For example, the following statement drops multiple columns:

```
ALTER TABLE dept80
DROP COLUMN ANNSAL,
DROP COLUMN hire_date;
```

# ALTER TABLE Statement: Add an Index or Constraint

Normally, you create indexes and constraints by using the CREATE TABLE statement when you create the table. Use the ALTER TABLE statement to add an index or constraint:

```
ALTER TABLE table
ADD PRIMARY KEY (column_name);
```

```
ALTER TABLE table
ADD [UNIQUE] INDEX (column_name);
```

```
ALTER TABLE table
ADD CONSTRAINT key_name
FOREIGN KEY (child_column_name)
REFERENCES parent_table (parent_column_name);
```

By using an ALTER TABLE statement, you can add a primary key (only if the table does not yet have a primary key, a UNIQUE index, a secondary index, or a foreign key. If you are adding a primary key or unique index to a table that contains data, the statement is rejected if the values are not unique. If you add a foreign key to a table that already contains data, if the values in the foreign key column in the child table do not have corresponding values in the primary key of the parent table, the statement is rejected.

## ALTER TABLE to Add a Constraint or Index: Example

The following example adds the constraint of `manager_id` being a foreign key in the `employees5` table that references the `employee_id` in the same table.

```
ALTER TABLE employees6
ADD CONSTRAINT emp6_manager_fk
FOREIGN KEY (manager_id)
REFERENCES employees6 (employee_id);
```

37

0

The statement in this slide creates a foreign key for the `manager_id` column in the `employees5` table that references the `employee_id` in the same table. Any `manager_id` must be the `employee_id` of some employee in the table.

# Creating Indexes by Using the CREATE INDEX Statement

With the CREATE INDEX statement, you can add a unique key or a secondary index with no constraints to a table.

Syntax:

```
CREATE [UNIQUE] INDEX index_name
ON table_name (column_name);
```

Example:

```
CREATE INDEX emp6_job_ix ON employees6 (job_id);
```

The example in this slide creates a secondary index for `job_id` in the `employees6` table. If you find that many queries use `job_id` as a search criteria, you might find it useful to add this index to speed up searches. Note that `job_id` in the `employees6` table cannot be a unique key because many people might have the same `job_id`. This statement is a shorter version of the following ALTER TABLE statement:

```
ALTER TABLE employees6
ADD INDEX emp6_job_ix (job_id);
```

# Viewing Index Definitions by Using the SHOW INDEX Statement

General syntax for viewing index definitions:

```
SHOW INDEX FROM table_name [FROM database_name]
```

Example:

```
SHOW INDEX FROM employees6;
```

#	Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible
1	employees6	0	PRIMARY	1	employee_id	A	1			NULL	BTREE			YES
2	employees6	0		1	email	A	1			NULL	BTREE			YES
3	employees6	1	emp6_name_ix	1	last_name	A	1			NULL	BTREE			YES
4	employees6	1	emp6_dept_fk	1	department_id	A	1			NULL	BTREE	YES		YES
5	employees6	1	emp6_manager_fk	1	manager_id	A	1			NULL	BTREE			YES
6	employees6	1	emp6_job_ix	1	job_id	A	1			NULL	BTREE			YES

39

0

If the database containing the table has been set as the default, you can omit the database name from the statement. In the output, the Non\_unique column displays a 1 if the index does not have the unique constraint and 0 if it is unique. The output displays the name of the index as well as the columns being indexed. The primary key always has the name PRIMARY. The Null column indicates if the column can be set to NULL.

# Showing How a Table Was Created with the SHOW CREATE TABLE Statement

View the statement that can be used to create the table. For example:

```
SHOW CREATE TABLE employees6;
```

```
CREATE TABLE `employees6` (
  `employee_id` int(11) NOT NULL AUTO_INCREMENT,
  `first_name` varchar(20) DEFAULT NULL,
  `last_name` varchar(25) NOT NULL,
  `email` varchar(25) NOT NULL,
  `phone_number` varchar(20) DEFAULT NULL,
  `hire_date` date NOT NULL,
  `job_id` varchar(10) NOT NULL,
  `salary` decimal(8,2) DEFAULT NULL,
  `commission_pct` decimal(2,2) DEFAULT NULL,
  `manager_id` int(11) DEFAULT '1',
  `department_id` int(11) DEFAULT '10',
  PRIMARY KEY(`employee_id`),
  UNIQUE KEY `email` (`email`),
  KEY `emp6_name_ix` (`last_name`),
  KEY `emp6_dept_fk` (`department_id`),
  KEY `emp6_manager_fk` (`manager_id`),
  KEY `emp6_job_ix` (`job_id`),
  CONSTRAINT `emp6_dept_fk` FOREIGN KEY (`department_id`) REFERENCES `departments` (`department_id`),
  CONSTRAINT `emp6_manager_fk` FOREIGN KEY (`manager_id`) REFERENCES `employees6` (`employee_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

40

O

The SHOW CREATE TABLE statement produces a statement that can create a table with the same properties as the table specified. This helps you to understand the structure of a table. You can use it to create a new table with the same structure. It does not show the exact statement that was used to create the table. It includes any modifications to the table from ALTER statements and might include more information, for example, about default values than were specified in any CREATE TABLE or ALTER TABLE statements. For example, the output shows default values of NULL for some columns even though that was not specified in any statement. The CREATE TABLE statement also includes some table options at the end even though those were not specified when creating the table.

**Note:** The results of the statement in MySQL Workbench might be too long to read in the Result Grid. Right-click the output and select **Open Value in Viewer**. Select the **Text** tab. You can save the generated CREATE TABLE statement to a file. When you are finished, click the **Close** button.

# Lesson Agenda

- Database objects
  - Naming rules
- **CREATE TABLE statement**
- Data types
- Indexes, keys, and constraints
- Column options
- Creating a table using a subquery
- ALTER TABLE statement
- DROP TABLE statement

41

0



For Instructor Use Only.  
This document should not be distributed.

# Dropping a Table

`DROP TABLE` removes the table definition and all its data entirely.

Syntax:

```
DROP TABLE [IF EXISTS] table_name [, table_name] ...;
```

The optional `IF EXISTS` clause prevents an error from occurring if the table does not exist. If it is omitted, dropping a table that does not exist generates an error.

Example:

```
DROP TABLE IF EXISTS dept80;
```



0

42

You can drop multiple tables in one statement. For example:

```
DROP TABLE IF EXISTS  
employees1, employees2, employees3, employees4, employees5;
```

# Summary

In this lesson, you should have learned how to use the CREATE TABLE, ALTER TABLE, and DROP TABLE statement to create a table, modify a table and columns, and include constraints.

- Categorize the main database objects
- Review the table structure
- List the data types that are available for columns
- Create a simple table
- Explain how constraints are created at the time of table creation



43

O

In this lesson, you should have learned the following:

## **CREATE TABLE**

- Use the CREATE TABLE statement to create a table and include constraints.
- Create a table based on another table by using a subquery.

## **DROP TABLE**

- Remove rows and a table structure.
- When executed, this statement cannot be rolled back.

## Practice 11b: Overview

This practice covers the following topics:

- Creating new tables
- Creating a new table by using the `CREATE TABLE AS` syntax
- Verifying that tables exist
- Altering tables
- Adding columns
- Dropping columns
- Dropping tables

44

0

You create new tables by using the `CREATE TABLE` statement and confirm that the new table was added to the database.

# Introduction to Data Dictionary Views

# Course Roadmap



2

0

In Unit 4, you get an introduction to data dictionary views. You learn how to create views on tables. You also learn about synonyms, sequences, and indexes, and how to create them.

For Instructor Use Only.  
This document should not be distributed.

# Objectives

After completing this lesson, you should be able to:

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

0

3

In this lesson, you are introduced to data dictionary views. You learn that 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

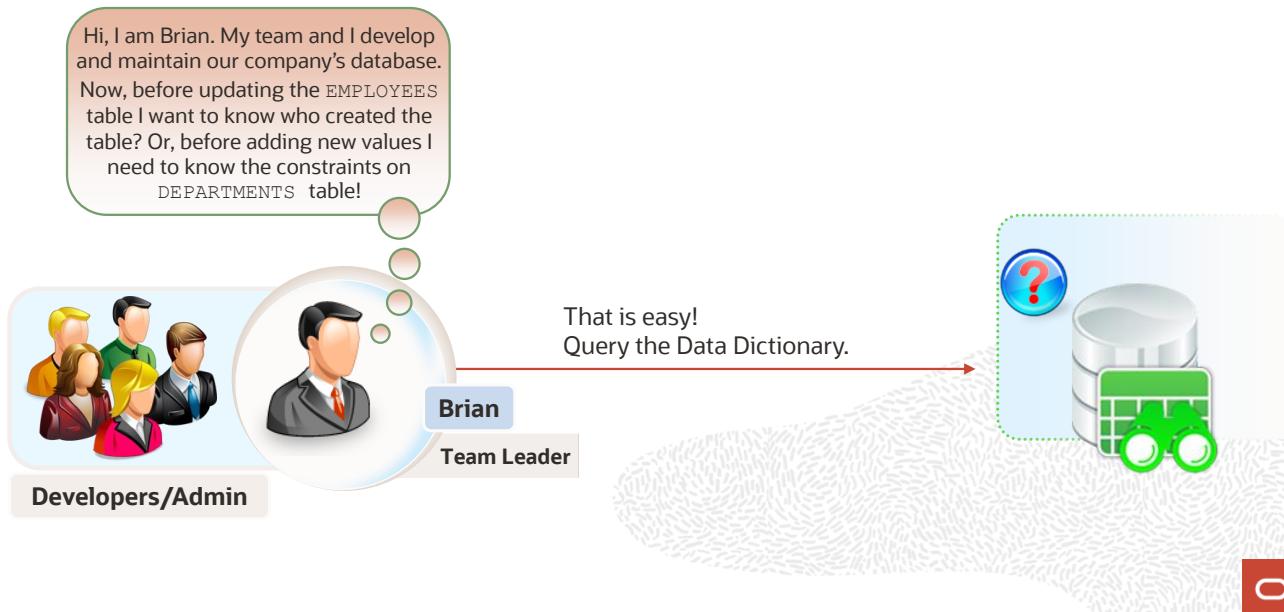
0

4

This section gives a brief introduction to data dictionary.

For Instructor Use Only.  
This document should not be distributed.

# Why Data Dictionary?



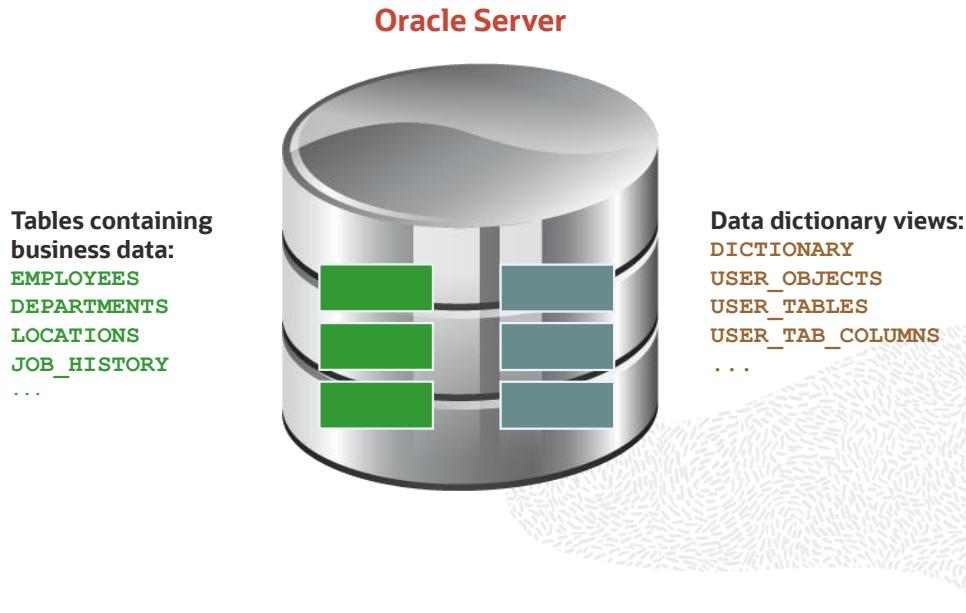
5

Brian and his team of developers are currently developing and maintaining his company's database. So, at some point in time, Brian (Team Leader) wants to alter the EMPLOYEES table by dropping/adding a column. Before altering the table, he wants to check with the owner of the table and discuss the effects of the change. He also wants to add a new department into the DEPARTMENTS table. Before inserting new values into the table, he needs to know the various constraints on the table so that they are not violated.

How do you think Brian will keep track of this information?

The good news is that Brian need not maintain any other document or table to store such information. All he needs to do is learn to query the data dictionary. The data dictionary is a list of in-built tables and views, which come along with the database. You learn more about data dictionary in the following slides.

# Data Dictionary



6

O

User tables are tables created by you and contain business data, such as EMPLOYEES. Along with user tables, 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.

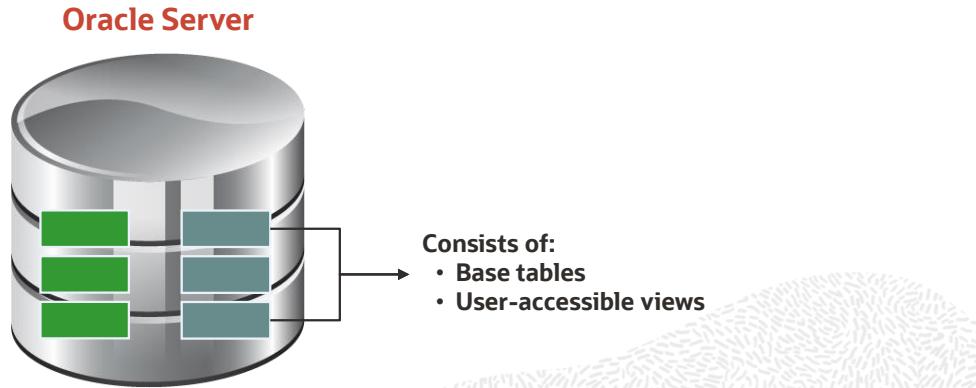
The data dictionary is structured in tables and views, just like other database data. The data dictionary is central to every Oracle database, and is an important tool for all users, from end users to application designers and database administrators.

You use SQL statements to access the data dictionary. Remember that the data dictionary is read-only and, therefore, you can issue queries only 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



7

O

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. You are mostly 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; doing so can compromise data integrity.

You will learn more about views and how to create them in the lesson titled “Creating Views.”

# Data Dictionary Structure

View naming convention is as follows:

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

8

0

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`, you will find 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 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
USER_OBJECTS	Objects owned by the user

9

0

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 in the slide, the `DICTIONARY` view is described. It has two columns. The `SELECT` statement in the slide 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*:

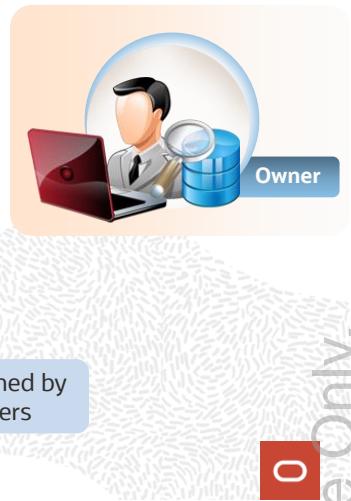
```
SELECT table_name
FROM   dictionary
WHERE  LOWER(comments) LIKE '%columns%';
```

**Note:** The table 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.

10

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 *Oracle® Database Reference 19c*.

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 LOC_COUNTRY_IX	INDEX	27-JUN-16	VALID
2 EMP_DEPARTMENT_IX	INDEX	27-JUN-16	VALID
3 LOC_STATE_PROVINCE_IX	INDEX	27-JUN-16	VALID
4 COUNTRY_C_ID_PK	INDEX	27-JUN-16	VALID
5 LOC_CITY_IX	INDEX	27-JUN-16	VALID
6 LOC_ID_PK	INDEX	27-JUN-16	VALID
7 JHIST_DEPARTMENT_IX	INDEX	27-JUN-16	VALID
8 JHIST_EMPLOYEE_IX	INDEX	27-JUN-16	VALID
9 DEPT_ID_PK	INDEX	27-JUN-16	VALID

...

→ Owned by the user

11

0

The example in the slide shows the name, type, date 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
- 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

0

12

This section discusses about querying the dictionary views for table information, column information, and constraint information.

For Instructor Use Only.  
This document should not be distributed.

# Table Information

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

13

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 *Oracle® Database Reference 19c*.

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

# Column Information

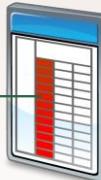
USER\_TAB\_COLUMNS:

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)

...

Get information about a column.



14

O

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, you will find detailed column information 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 19c*.

# 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

15

0

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.

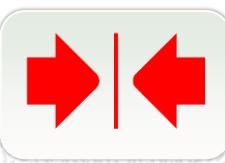
# Constraint Information

- `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
```

Name	Nu11	Type
OWNER		VARCHAR2(128)
CONSTRAINT_NAME	NOT NULL	VARCHAR2(128)
CONSTRAINT_TYPE		VARCHAR2(1)
TABLE_NAME	NOT NULL	VARCHAR2(128)
SEARCH_CONDITION	LONG	
SEARCH_CONDITION_VC		VARCHAR2(4000)
R_OWNER		VARCHAR2(128)
R_CONSTRAINT_NAME		VARCHAR2(128)
DELETE_RULE		VARCHAR2(9)
STATUS		VARCHAR2(8)

...



0

16

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 *Oracle® Database Reference 19c*.

# 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

17

0

In the example in the slide, 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 inactive.

# 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

18

0

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

0

19

In this section, you learn how to add a comment to a table and how to view the comment by using dictionary views.

For Instructor Use Only.  
This document should not be distributed.

# 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`

20

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 '';
```

# Summary

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

- DICTIONARY
- USER\_OBJECTS
- USER\_TABLES
- USER\_TAB\_COLUMNS
- USER\_CONSTRAINTS
- USER\_CONS\_COLUMNS

21



O

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 12: 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



0

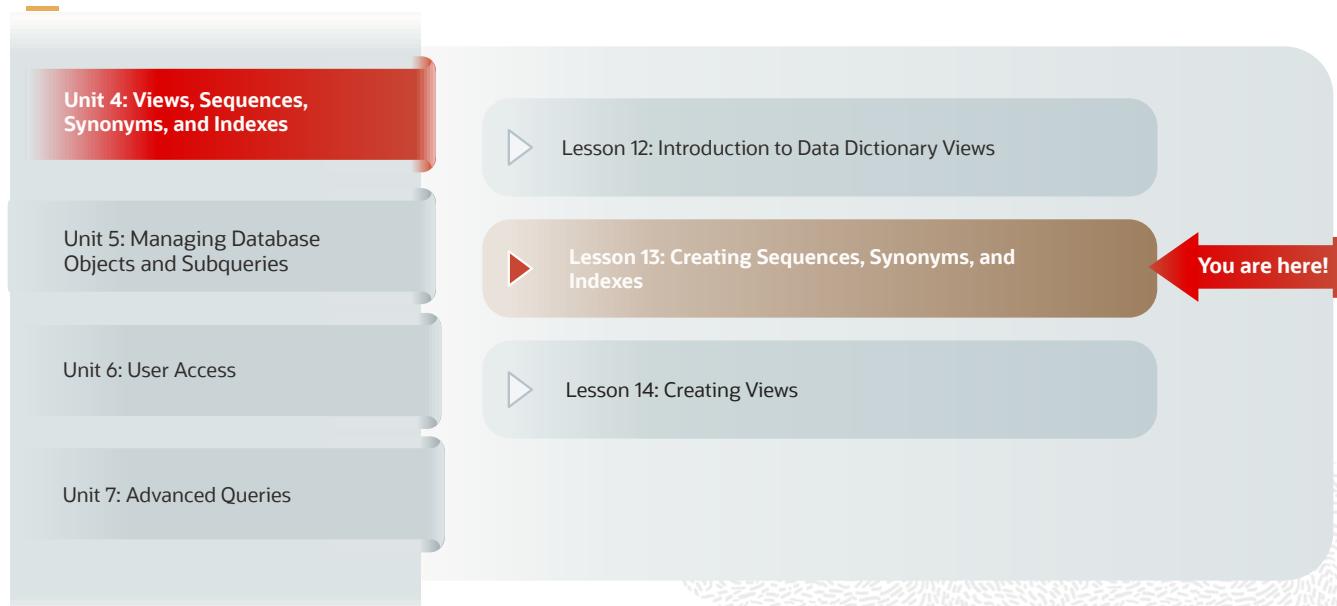
22

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

For Instructor Use Only.  
This document should not be distributed.

# Creating Sequences, Synonyms, and Indexes

# Course Roadmap



2

0

In Unit 4, you get an introduction to data dictionary views. You learn how to create views on tables. You also learn about synonyms, sequences, indexes, and how to create them.

For Instructor Use Only.  
This document should not be distributed.

# 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



O

3

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 and 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

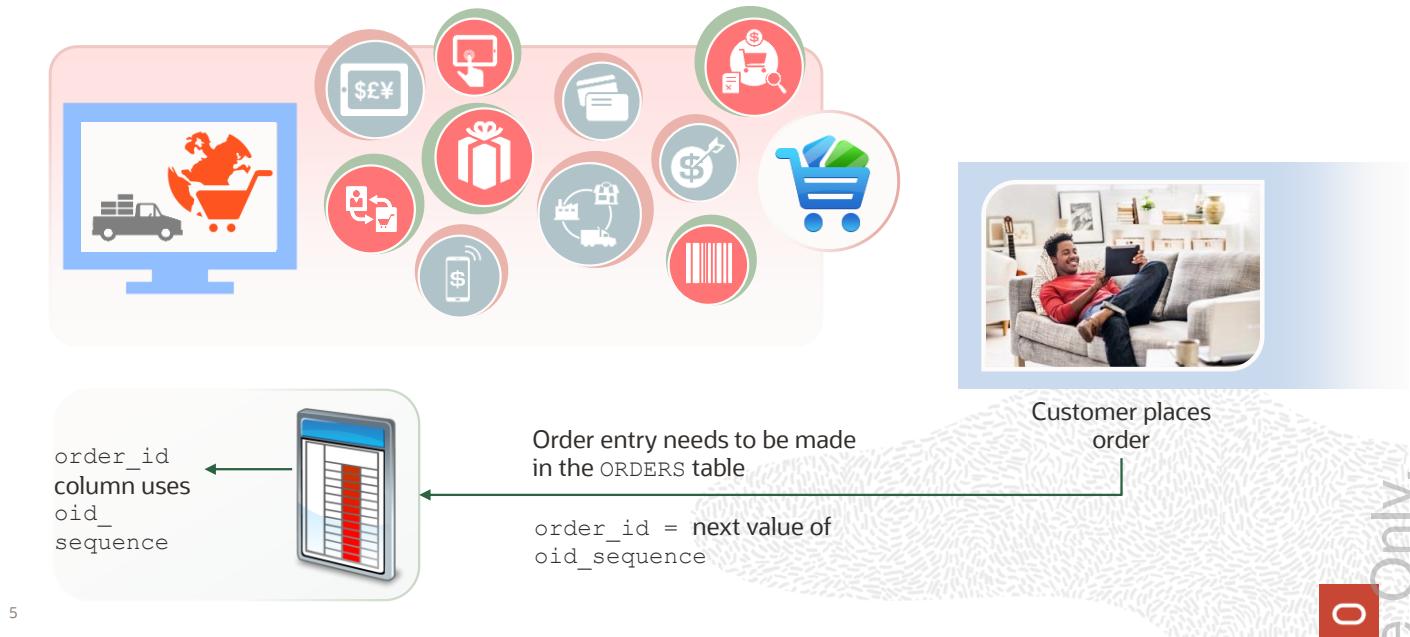
4

O



This section discusses the overview of sequences. You will learn how to create, modify, and use sequences. You will also learn about the NEXTVAL and CURRVAL pseudocolumns and how to use them in creating default values for a column.

# E-Commerce Scenario



5

0

OracleKart is an online e-commerce company selling a variety of goods. The back-end database maintains a table for all orders placed by the customers. Each order requires a unique ID to store the order information correctly. How do you think OracleKart will generate the `order_id` in the `ORDERS` table? Should an administrator manually insert a unique `order_id` each time an order is placed?

Brian, the DBA, is aware that manually generating unique IDs is not efficient. An error can cause incorrect data mapping, which will lead to incorrect order delivery. For data integrity and work efficiency, Brian can make use of SQL sequence to generate order IDs automatically.

To solve scenarios like the above, SQL provides sequences. Using a sequence, you can generate a unique sequence of numbers according to your need. Let us look into some more advantages of a sequence and learn how to use it in a SQL query.

# 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

6

0

As shown in the slide, 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.

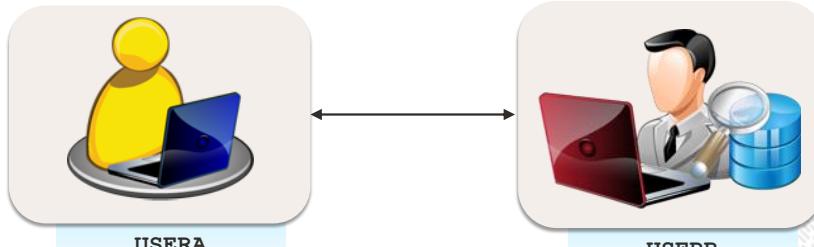
If you want to improve the performance of data retrieval queries, you should consider creating an **index**.

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;
```

7

## What is a schema?

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;
```

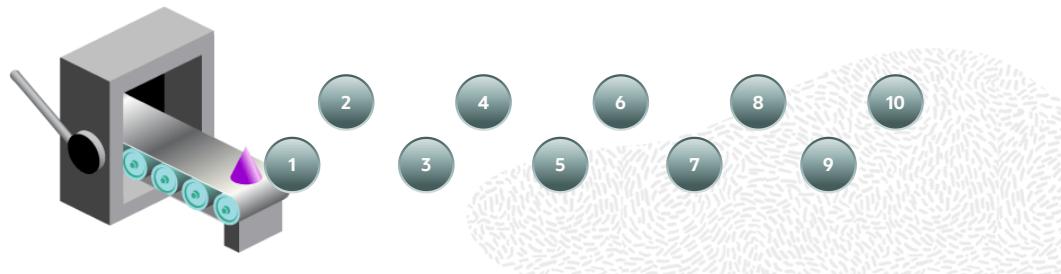
If **USERB** wants to access the **EMPLOYEES** table that is owned by **USERA**, **USERB** must prefix the table name with the schema name:

```
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



8

0

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 }
| { SCALE | NOSCALE }
| { SHARD | NOSHARD }
];
```



9

O

Automatically generate sequential numbers by using the CREATE SEQUENCE statement.

In the syntax:

<i>Sequence</i>	Is the name of the sequence generator
START WITH <i>n</i>	Specifies the first sequence number to be generated (If this clause is omitted, the sequence starts with 1.)
INCREMENT BY <i>n</i>	Specifies the interval between sequence numbers, where <i>n</i> is an integer (If this clause is omitted, the sequence increments by 1.)
MAXVALUE <i>n</i>	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 <i>n</i>	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	If specified, guarantees that sequence numbers are generated in order of request. This clause is useful if you are using the sequence numbers as timestamps.
NOORDER	If specified, sequence numbers are not 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 <i>n</i>   NOCACHE	Specifies how many values the Oracle Server pre-allocates and keeps in memory (By default, the Oracle server caches 20 values.)
SCALE [EXTEND NOEXTEND]   NOSCALE	If specified, a numeric offset is affixed to the beginning of the sequence which removes all duplicates in generated values (NOSCALE is the default option).

If EXTEND is specified with SCALE, the generated sequence values are all of length (x+y), where x is the length of the scalable offset (default value is 6), and y is the maximum number of digits in the sequence (maxvalue/minvalue) (NOEXTEND is the default).

SHARD [EXTEND | NOEXTEND] | NOSHARD If specified, the sequence generates unique sequence numbers across shards. If EXTEND is specified with the SHARD clause, the generated sequence values are all of length (x = y), where x is the length of a SHARD offset of size 4. (NOEXTEND is the default).

# 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
  SCALE;
```

Sequence DEPT\_DEPTID\_SEQ created.

11

0

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 and adds a 6 digit prefix to the sequence number (three digit instance number followed by a three digit session number). It does not allow caching, and does not cycle.

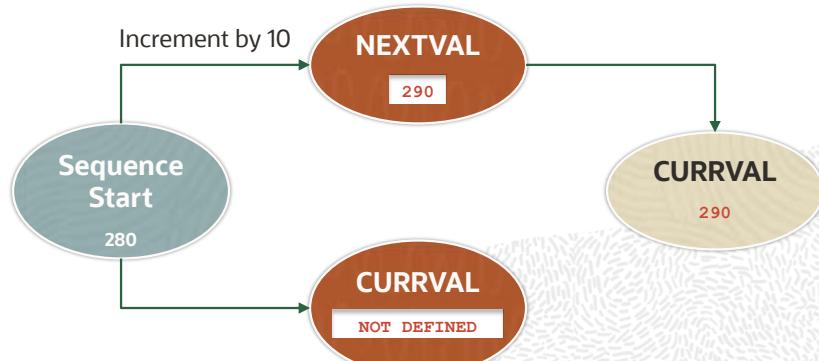
Do not use the CYCLE option if the sequence is used to generate primary key values, unless you have a reliable mechanism that purges old rows faster than the sequence cycles.

For more information, see the “CREATE SEQUENCE” section in *Oracle Database SQL Language Reference* for Oracle Database 19c.

**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 can be referenced.



12

O

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` clause
- 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 19c.

# 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 row inserted.
```

- View the current value for the DEPT\_DEPTID\_SEQ sequence:

```
SELECT dept_deptid_seq.CURRVAL
FROMdual;
```

CURRVAL
1 280

14

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 shown in the slide.

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 EMPLOYEES\_SEQ has already been created to generate new employee numbers.

# SQL Column Defaulting Using a Sequence

- You can use the SQL syntax <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 ID_SEQ START WITH 1;
CREATE TABLE emp (ID NUMBER DEFAULT ID_SEQ.NEXTVAL NOT NULL,
                  name VARCHAR2(10));
INSERT INTO emp (name) VALUES ('john');
INSERT INTO emp (name) VALUES ('mark');
SELECT * FROM emp;
```

Sequence ID\_SEQ created.  
Table EMP created.  
1 row inserted.  
1 row inserted.  
ID NAME  
-----  
1 john  
2 mark

15

O

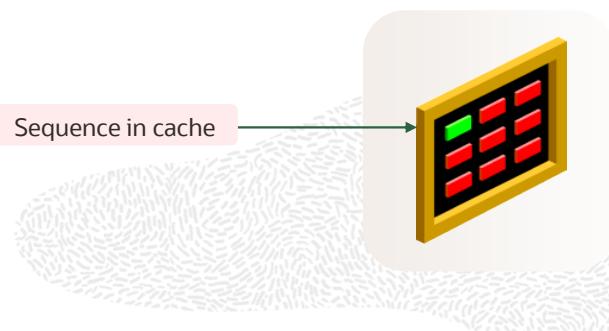
SQL syntax for column defaults has been enhanced so that it allows <sequence>.nextval, <sequence>.currval as a SQL column defaulting expression for numeric columns, where <sequence> is an Oracle database sequence.

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 ID\_SEQ 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



16

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

The sequence generators generally issue sequential numbers without gaps and 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
    NOSCALE;
```

```
Sequence DEPT_DEPTID_SEQ altered.
```

17

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}]
    [{SCALE [EXTEND | NOEXTEND] | NOSCALE}]
    [{SHARD [EXTEND | NOEXTEND] | NOSHARD};
```

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 19c.

# 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.
```

18

O

- 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 by 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
    NOSCALE;
```

```
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
```

SEQUENCE_NAME	VARCHAR2(128)
MIN_VALUE	NUMBER
MAX_VALUE	NUMBER
INCREMENT_BY	NUMBER
CYCLE_FLAG	VARCHAR2(1)
ORDER_FLAG	VARCHAR2(1)
CACHE_SIZE	NUMBER
LAST_NUMBER	NUMBER
SCALE_FLAG	VARCHAR2(1)
EXTEND_FLAG	VARCHAR2(1)
SHARDED_FLAG	VARCHAR2(1)
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 following are the columns in this view:

- 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. If NOCACHE is specified, the `LAST_NUMBER` column displays the next available sequence number
- SCALE\_FLAG:** Indicates whether this is a scalable sequence (Y) or not (N)
- EXTEND\_FLAG:** Indicates whether this scalable sequence's generated values extend beyond MAX\_VALUE or MIN\_VALUE (Y) or not (N)
- SHARDED\_FLAG:** Indicates whether this is a sharded sequence (Y) or not (N)
- SESSION\_FLAG:** Whether the order value is session private
- KEEP\_VALUE:** Response time after an error

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 and 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

20

0

This section provides an overview of synonyms. You also learn how to create and drop synonyms.

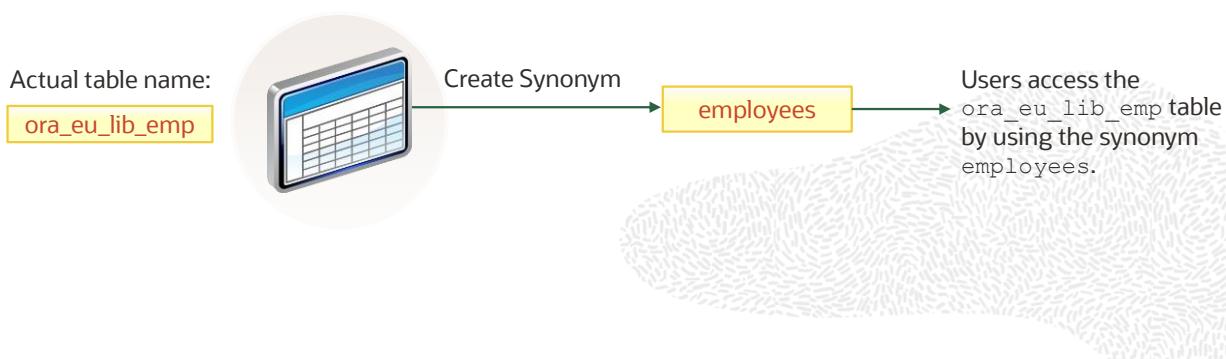


For Instructor Use Only.  
This document should not be distributed.

# Synonyms

A synonym:

- Is a database object
- Can be created to give an alternative name to a table or to another 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



21

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

You can create synonyms to give an alternative name to a table or to another 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

- You can 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;
```

22

O

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
<i>synonym</i>	Is the name of the synonym to be created
<i>object</i>	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 19c.

# Creating and Removing Synonyms

- Create a shortened name for the DEPARTMENTS table:

```
CREATE SYNONYM dept  
FOR departments;
```

Synonym DEPT created.

- Drop a synonym:

```
DROP SYNONYM dept;
```

Synonym DEPT dropped.

23

## Creating a Synonym

The example in the slide creates a synonym for the DEPARTMENTS table 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 19c.

# Synonym Information

```
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)
ORIGIN_CON_ID		NUMBER

```
SELECT *  
FROM user_synonyms;
```

SYNONYM_NAME	TABLE_OWNER	TABLE_NAME	DB_LINK	ORIGIN_CON_ID
DEPT	TEACH_B	DEPARTMENTS	(null)	3

24

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)
- `ORIGIN_CON_ID`: The container ID of the PDB where the data originates. In this case, it is 3.

# 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 and 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

25

O

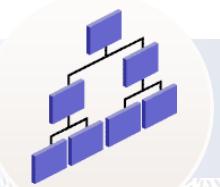


This section gives an overview of indexes. We will discuss how to create different types of indexes and how to remove indexes.

# 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



O

26

You can use an index to speed up the retrieval of rows by using a pointer and improve 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 deleted 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 19c*.

# 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 a unique or nonunique index on columns to speed up access to the rows.



27

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.
```

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) on 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 `rowid` 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 19c.

## 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
EMP_ID_IDX	NEW_EMP

29

0

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

1 row fetched in 0.012 seconds

\*Note: The time may differ for you.

31

O

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; however, 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.

Observe that each time you run the same query on an index based table, the time required to fetch the row(s) reduces.

## Creating Multiple Indexes on the Same Set of Columns

You can create multiple indexes 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 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.

## Creating Multiple Indexes on the Same Set of Columns: Example

```
CREATE INDEX emp_id_name_ix1  
ON employees(employee_id, first_name);  
Index EMP_ID_NAME_IX1 created.
```

1

```
ALTER INDEX emp_id_name_ix1 INVISIBLE;  
Index EMP_ID_NAME_IX1 altered.
```

2

```
CREATE BITMAP INDEX emp_id_name_ix2  
ON employees(employee_id, first_name);  
Bitmap index EMP_ID_NAME_IX2 created.
```

3

33

0

1. 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. By default, Oracle Server creates a B-tree index. In a B-tree, you traverse the branches until you get to the node that has the data you are searching for.
2. After the creation of the index, it is altered to make it invisible.
3. 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
```

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)
COMPRESSION		VARCHAR2(13)
PREFIX_LENGTH		NUMBER
TABLESPACE_NAME		VARCHAR2(30)

...

34



0

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 *Oracle® Database Reference 19c*.

## USER\_INDEXES: Examples

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

1

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

...

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

2

INDEX_NAME	TABLE_NAME
1 SYS_C0010979	EMP_LIB

In example 1 in the slide, 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 example 2 in the slide, 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
```

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

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 example in the slide, 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:

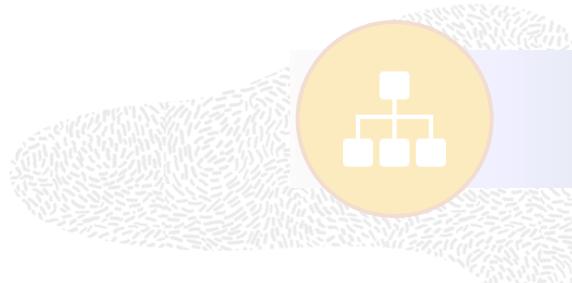
```
DROP INDEX index;
```

- 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.



37

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

You can 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 by using the `ONLINE` keyword:

```
DROP INDEX emp_indx 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.

# 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\_INDEXES
  - USER\_SEQUENCES
  - USER\_SYNONYMS

38



O

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

# Practice 13: 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



0

39

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.

For Instructor Use Only.  
This document should not be distributed.

# Creating Views

# Course Roadmap

## Unit 4: Views, Sequences, Synonyms, and Indexes

▶ Lesson 12: Introduction to Data Dictionary Views

## Unit 5: Managing Database Objects and Subqueries

▶ Lesson 13: Creating Sequences, Synonyms, and Indexes

## Unit 6: User Access

▶ Lesson 14: Creating Views

## Unit 7: Advanced Queries

You are here!

0

2

In Unit 4, you will get an introduction to data dictionary views. You will learn how to create views on tables. You will also learn about synonyms, sequences, indexes, and how to create them.

For Instructor Use Only.  
This document should not be distributed.

# Objectives

After completing this lesson, you should be able to:

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

0

3

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

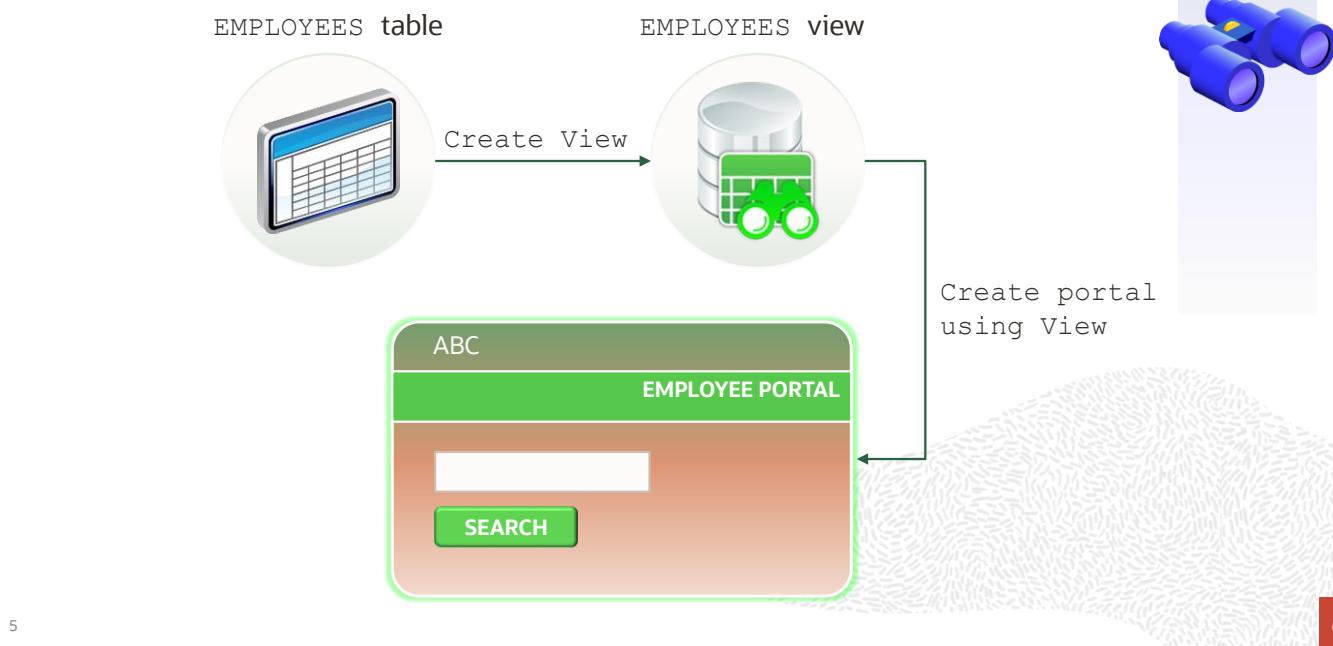
4

0

This section gives an overview of views.

For Instructor Use Only.  
This document should not be distributed.

# Why Views?



ABC is a company, which consists of around 1000 employees. The HR department of the company maintains a database to store all the employee information, such as full name, date of birth, department, salary, manager, designation, phone number, hire date, and so on. This information is confidential and cannot be accessed by all the employees.

The company now decides to create an employee search portal for the use of employees. This portal should display only the necessary information (such as name, department, and manager) and should *not* display any confidential information (such as salary). To achieve this, the company need not create a separate table, which stores the same data with fewer columns.

Instead, a view is created with the columns required based on the underlying `EMPLOYEES` table. This helps in removing redundancy and hiding sensitive information securely. You learn more about views and how to use them in the following slides.

# 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

6

0

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							
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
100	Steven	King	SKING	515.123.4567	17-JUN-11	AD_PRES	24000
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-09	AD_VP	17000
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-09	AD_VP	17000
103	Alexander				18-AN-14	IT_PROG	9000
104	Bruce				18-AY-15	IT_PROG	6000
105	David				18-BN-13	IT_PROG	4800
106	Eduardo				18-EB-14	IT_PROG	4800
107	Gloria				18-EB-15	IT_PROG	4200
108	Harley				18-EB-16	FI_MGR	12008
109	James				18-EB-17	FI_ACCOUNT	9000
110	Jeffrey				18-EB-18	FI_ACCOUNT	8200
111	Klaus				18-EB-19	FI_ACCOUNT	7700
112	Lex				18-EB-20	FI_ACCOUNT	7800
113	Luis	Popp	LPUPP	515.124.4567	07-DEC-15	FI_ACCOUNT	6900
114	Den	Raphaely	DRAPHEAL	515.127.4561	07-DEC-10	PU_MAN	11000
115	Alexander	Khoo	AKHOO	515.127.4562	18-MAY-11	PU_CLERK	3100

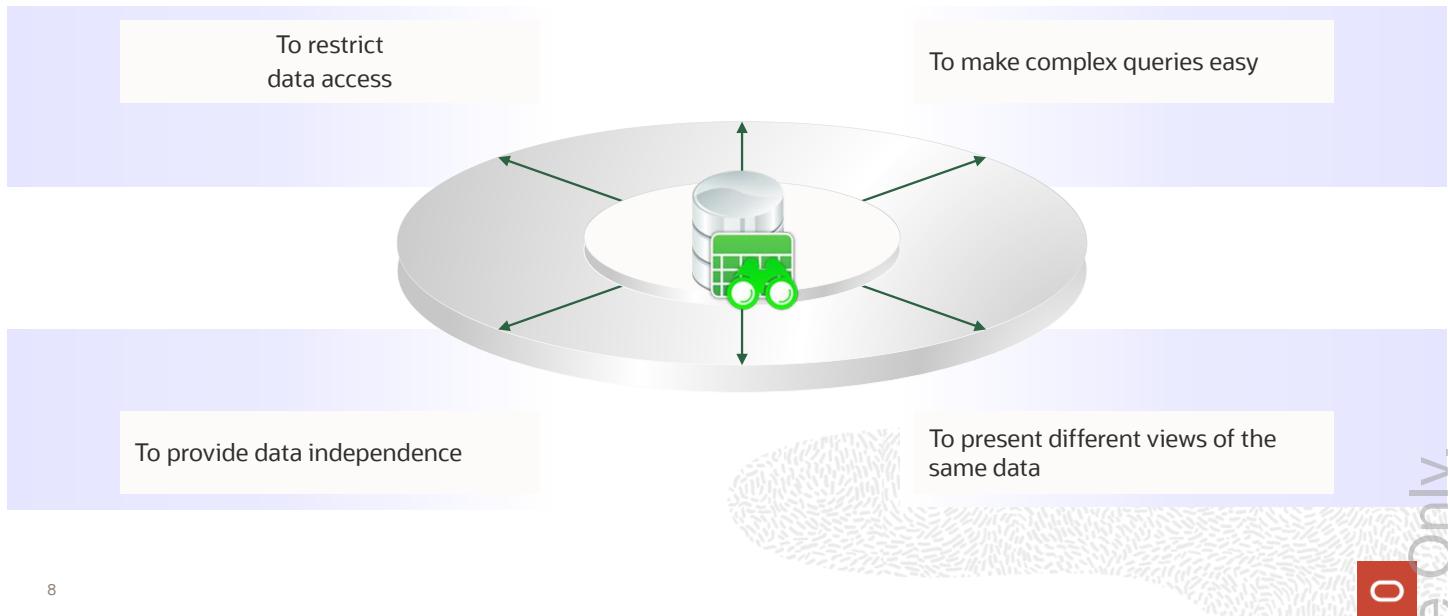
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY
100	Steven	King	24000
101	Neena	Kochhar	17000
102	Lex	De Haan	17000
103	Alexander	Hunold	9000
104	Bruce	Ernst	6000
113	Luis	Popp	515.124.4567
114	Den	Raphaely	515.127.4561
115	Alexander	Khoo	515.127.4562

7

0

You can present logical subsets or combinations of data by creating views of tables. A view is a schema object and is stored as a `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



8

The following are some of the advantages of views:

- Views restrict access to 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 19c.

# 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

9

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:
  - Derives data from only one table
  - Contains no functions or groups of data
  - Usually performs DML operations through the view
- A complex view:
  - 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

10

0

In this section, you learn how to create, modify, and retrieve data from a view.

For Instructor Use Only.  
This document should not be distributed.

# 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.



0

11

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.

Creates the view regardless of whether or not the base tables exist

Creates the view only if the base tables exist (This is the default.)

Is the name of the view

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;
```

12

0

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.

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 re-granting 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.

```
SELECT ID_NUMBER, NAME, ANN_SALARY
  FROM salvu50;
```

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



0

14

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.

For Instructor Use Only.  
This document should not be distributed.

# 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 re-granting 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.
```

16

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	Nu11	Type
VIEW_NAME	NOT NULL	VARCHAR2(128)
TEXT_LENGTH		NUMBER
TEXT		LONG
TEXT_VC		VARCHAR2(4000)
TYPE_TEXT_LENGTH		NUMBER
TYPE_TEXT		VARCHAR2(4000)

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_id WITH READ ONLY

17

0

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

18

0

In this section, you will learn how to perform DML operations on a view.

For Instructor Use Only.  
This document should not be distributed.

# 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



O

19

- You can perform DML operations on data in 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 Modify 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

20

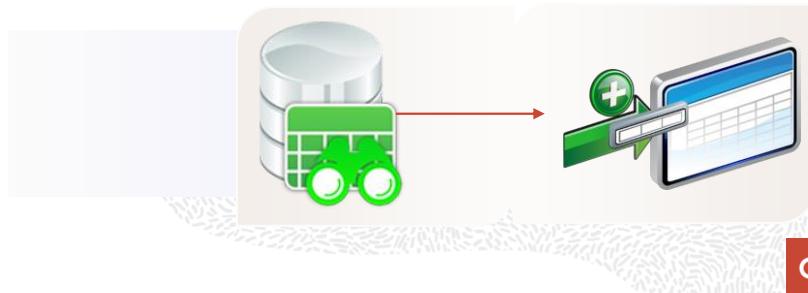


You can modify data in a view unless it contains any of the conditions mentioned in the slide.

## Rules for Performing Insert Operations Through a View

You cannot add data in 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



21

O

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

# 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.

22

O

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 on your view 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.

23



O

The example in the following 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.
```

24

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

25

0

In this section, you learn how to drop a view.

For Instructor Use Only.  
This document should not be distributed.

# Removing a View

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

Syntax:

```
DROP VIEW view;
```

Example:

```
DROP VIEW empvu80;
```

```
View EMPVU80 dropped.
```



X

O

26

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.



# Summary

In this lesson, you should have learned how to:

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

0



27

In this lesson, you should have learned about views.

For Instructor Use Only.  
This document should not be distributed.

# Practice 14: 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



0

28

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

For Instructor Use Only.  
This document should not be distributed.

# Managing Schema Objects

For Instructor Use Only.  
This document should not be distributed.

# Course Roadmap



In Unit 5, you will learn to use SQL statements to query and display data from multiple tables using Joins, use subqueries when the condition is unknown, use group functions to aggregate data, and use set operators.

0

For Instructor Use Only.  
This document should not be distributed.

# Objectives

After completing this lesson, you should be able to:

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



O

3

In this lesson, you learn about constraints and altering existing objects. You will 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

0

4

This section discusses how to manage constraints. You learn to add, drop, enable, disable, and defer a constraint.



For Instructor Use Only.  
This document should not be distributed.

# 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>);
```

5

0

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

In the syntax:

<code>table_name</code>	Is the name of the table
<code>constraint_name</code>	Is the name of the constraint
<code>type</code>	Is the constraint type
<code>column_name</code>	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.

6

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.

**Note:** The EMP2 table is created with the following statement:

```
CREATE TABLE emp2 as  
SELECT * FROM employees;
```

# 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 EMP2 table and drop the associated FOREIGN KEY constraint on the EMP2.MANAGER\_ID column:

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

Table EMP2 altered.

7

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:

<i>table</i>	Is the name of the table
<i>column</i>	Is the name of the column affected by the constraint
<i>constraint</i>	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.

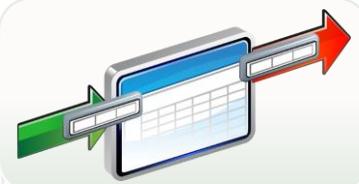
# Dropping a Constraint ONLINE

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;
```

```
Table MYEMP2 altered.
```

DML operations allowed while  
dropping a constraint



8

0

You can also drop a constraint by 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 are allowed on the table while dropping the constraint.

**Note:** The `myemp2` table is created using the following statement:

```
CREATE TABLE myemp2
(emp_id NUMBER(6) CONSTRAINT emp_name_pk PRIMARY KEY ,
emp_name VARCHAR2(20));
```

# 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.

0

9

## 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 clause 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 clause 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 clause 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

10

0

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);  
ERROR: col1\_pk is a parent key.
- ALTER TABLE test1 DROP (col1);  
ERROR: 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) CASCADE CONSTRAINTS;
```

Table TEST1 altered.

11

0

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);
```

## Guidelines

- Enabling a PRIMARY KEY constraint that was disabled with the CASCADE option does not enable any FOREIGN KEYS that are dependent on 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.

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

1

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

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

2

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

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

3

12

0

The RENAME table clause enables you to rename an existing table in any schema (except the SYS schema). 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 examples in the slide 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 1 shows that the marketing table is renamed new\_marketing.

Example 2 shows that the team\_id column of the new\_marketing table is renamed id.

Example 3 shows that the mktg\_pk constraint 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. It will also disable all dependent FOREIGN KEY constraints automatically.

```
ALTER TABLE    emp2  
DISABLE CONSTRAINT emp_dt_fk;
```

Table EMP2 altered.

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

Table DEPT2 altered.

13

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 by using the CASCADE option.

## Syntax

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

In the syntax:

<i>table</i>	Is the name of the table
<i>constraint</i>	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.

14

0

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:

<i>table</i>	Is the name of the table
<i>constraint</i>	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.

15

O

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:** The 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.

17

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 example in the slide:

- 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 (TEACH\_B.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);
```

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 (TEACH_B.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.

```
ALTER SESSION SET CONSTRAINTS = DEFERRED;
```

```
Session altered.
```

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 row 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 (TEACH_B.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.

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

```
Session altered.
```

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 (TEACH_B.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:

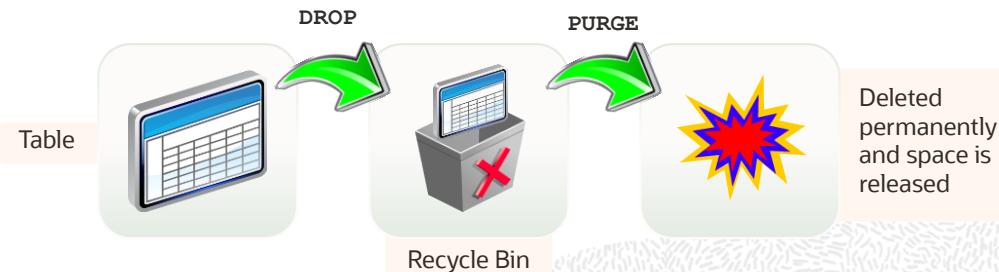
```
SET CONSTRAINT newemp_det_pk DEFERRED;
```

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

## DROP TABLE ... PURGE

```
DROP TABLE emp_new_sal PURGE;
```

```
Table EMP_NEW_SAL dropped.
```



19

O

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, then 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

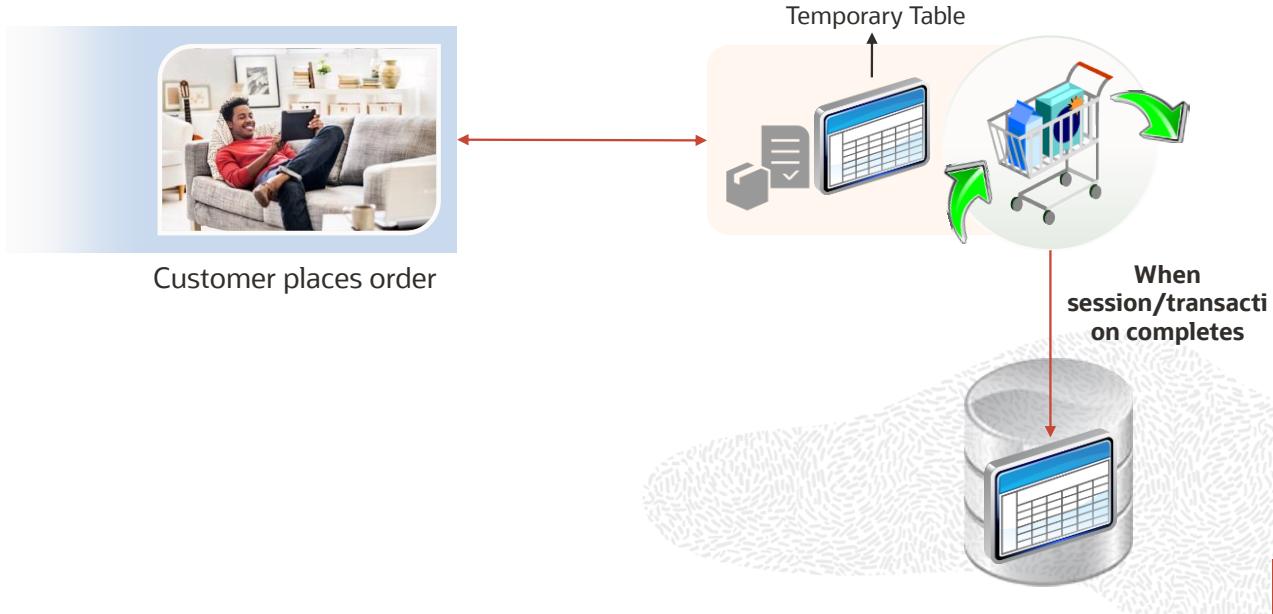
20

0

In this section, you learn to create and use temporary tables.

For Instructor Use Only.  
This document should not be distributed.

# Using Temporary Tables



21

O

Brian is designing the OracleKart application. When customers place an order, the orders are to be stored in an `ORDER_DETAILS` table. Brian wants to enable customers, before they finalize their order, to use a “shopping cart” in which they can list items they wish to purchase. They should be able to add and remove items from this shopping cart. Storing this information in the `ORDER_DETAILS` table is not feasible because this table is not accessible to customers. Moreover, this data is not required to be stored permanently but only for the duration of a transaction. SQL enables you to store such information in temporary tables.

The shopping cart can be a temporary table. Each item is represented by a row in the temporary table. James continues to add and remove items from his cart before he finalizes on the items to buy. During this session, the cart data is private and the data in the temporary table keeps changing. After James finalizes his shopping and makes the payment, the application moves the rows from his cart to a permanent table, such as the `ORDER_DETAILS` table. At the end of the session, the data in the temporary table is automatically dropped.

Therefore, temporary tables are useful in applications where a result set must be buffered.

# Temporary Table

A temporary table :

- holds data that exists only for the duration of a transaction or session.
- data is private to the session.
- can be either a Global Temporary Table or a Private Temporary Table.

22

0

A temporary table holds data that exists only for the duration of a transaction or session.

Data in a temporary table is private to the session. Each session can only see and modify its own data.

You can create either a global temporary table or a private temporary table.

# Temporary Table Characteristics

Characteristic	Global	Private
Naming rules	Same as for permanent tables	Must be prefixed with ORA\$PTT_
Visibility of table definition	All sessions	Only the session that created the table
Storage of table definition	Disk	Memory only
Types	Transaction-specific (ON COMMIT DELETE ROWS) or session-specific (ON COMMIT PRESERVE ROWS)	Transaction-specific (ON COMMIT DROP DEFINITION) or session-specific (ON COMMIT PRESERVE DEFINITION)

# Creating a Global Temporary Table

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

1

```
CREATE GLOBAL TEMPORARY TABLE emp_details
ON COMMIT PRESERVE ROWS;
```

2

0

24

The definition of a global temporary table is visible to all sessions, but the data in a global temporary table is visible only to the session that inserts the data into the table.

To create a global temporary table, use the following command:

```
CREATE GLOBAL TEMPORARY TABLE tablename(
  column_definition,
  ...
  ON COMMIT [PRESERVE | DELETE] ROWS;
```

The **ON COMMIT** clause indicates if the data in the table is transaction-specific (the default) or session-specific, the implications of which are:

**PRESERVE ROWS:** This creates a global temporary table that is session specific. A session gets bound to the global temporary table with the first insert into the table in the session. 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.

**DELETE ROWS:** This creates a global temporary table that is transaction specific. A session becomes bound to the global temporary table with a transactions 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.

When you create a global temporary table in an Oracle database, you create a static table definition. Like permanent tables, global temporary tables are defined in the data dictionary. However, global 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.

# Creating a Private Temporary Table

```
CREATE PRIVATE TEMPORARY TABLE ORA$PTT_sales_trans
  (time_id      DATE,
   amt_sold    NUMBER(8,2))
ON COMMIT DROP DEFINITION;
```

1

```
CREATE PRIVATE TEMPORARY TABLE ORA$PTT_sales_sess
  (time_id      DATE,
   amt_sold    NUMBER(8,2))
ON COMMIT PRESERVE DEFINITION;
```

2

25

Private temporary tables are temporary database objects that are dropped at the end of a transaction or session. Private temporary tables are stored in memory and the metadata and content of a private temporary table is visible only within the session that created it.

To create a private temporary table, use the following command:

```
CREATE PRIVATE TEMPORARY TABLE tablename(  

  column_definition,  

  ...)  
ON COMMIT [DROP | PRESERVE] DEFINITION;
```

The **ON COMMIT** clause indicates if the data in the table is transaction-specific (the default) or session-specific, the implications of which are as follows:

**DROP DEFINITION:** This creates a private temporary table that is transaction specific. All data in the table is lost, and the table is dropped at the end of transaction.

**PRESERVE DEFINITION:** This creates a private temporary table that is session specific. All data in the table is lost, and the table is dropped at the end of the session that created the table.

**Note:** Names of private temporary tables must be prefixed according to the initialization parameter PRIVATE\_TEMP\_TABLE\_PREFIX.

# 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

0

For Instructor Use Only.  
This document should not be distributed.

In this section, you learn to create and use external tables.

# External Tables



27

O

In an external table, you store the metadata in the database and the actual data outside the database. This external table 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:

- The `ORACLE_LOADER` access driver 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 rows from a `SELECT` statement to be loaded into an external table are written as part of a `CREATE TABLE ... ORGANIZATION EXTERNAL...AS SELECT` statement by the access driver. 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;
```

28

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 it 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 for access 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 directory AS 'path_name';
```

In the syntax:

OR REPLACE

creating,

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

Syntax:

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

30

0

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, `<locationSpecifier>` 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.

32

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 file is located at  
`/home/oracle/labs/sql2/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 '/home/oracle/labs/sql2/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:

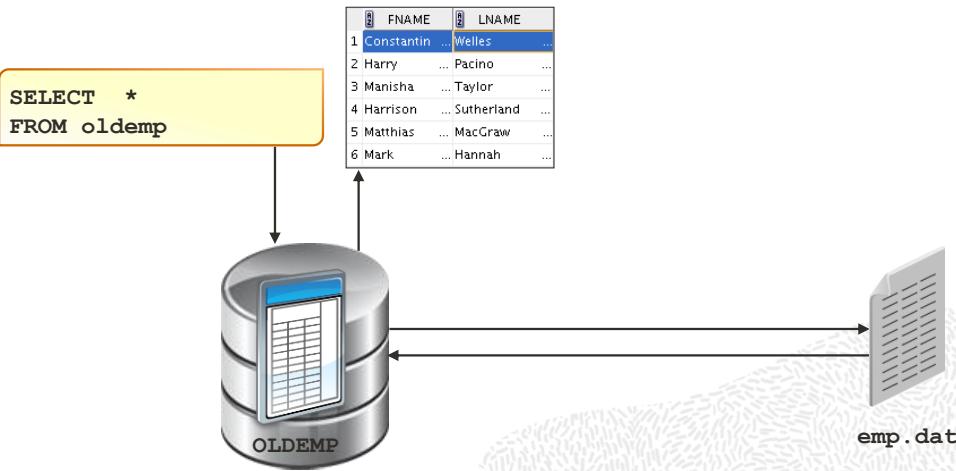
```
/home/oracle/labs/sql2/emp_dir/emp.dat
```

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.

**Note:** The `emp_dir` directory on the file system must have write permission for user and others for the external table to work successfully.

# Querying External Tables



33

An external table does not tell you how data is stored in the database. It also does not tell you how data is stored in the external source. Instead, it tells you 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.

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.

34

O

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 OR REPLACE DIRECTORY emp_dir AS '/stage/Labs/sql12/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;
```

# Summary

In this lesson, you should have learned how to:

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

35



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 15: Overview

This practice covers the following topics:

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

36

0

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



For Instructor Use Only.  
This document should not be distributed.