

---

# Oracle Database 11*g*: SQL Fundamentals I

Electronic Presentation

---

D49996GC11  
Edition 1.1  
April 2009

**ORACLE®**

## **Authors**

Puja Singh  
Brian Pottle

## **Technical Contributors and Reviewers**

Claire Bennett  
Tom Best  
Purjanti Chang  
Ken Cooper  
László Czinkóczy  
Burt Demchick  
Mark Fleming  
Gerlinde Frenzen  
Nancy Greenberg  
Chaitanya Koratamaddi  
Wendy Lo  
Timothy Mcglue  
Alan Paulson  
Bryan Roberts  
Abhishek Singh  
Lori Tritz  
Michael Versaci  
Lex van der Werff

## **Editors**

Amitha Narayan  
Vijayalakshmi Narasimhan  
Raj Kumar

## **Graphic Designer**

Satish Bettegowda

## **Publishers**

Sujatha Nagendra  
Syed Ali

**Copyright © 2009, Oracle. All rights reserved.**

## **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. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

## **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 RIGHTS**

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

## **Trademark Notice**

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

# I Introduction

# Lesson Objectives

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

- Understand the goals of the course
- List the features of Oracle Database 11g
- Discuss the theoretical and physical aspects of a relational database
- Describe Oracle server's implementation of RDBMS and object relational database management system (ORDBMS)
- Identify the development environments that can be used for this course
- Describe the database and schema used in this course

# Lesson Agenda

- Course objectives, agenda, and appendixes used in the course
- Overview of Oracle Database 11g and related products
- Overview of relational database management concepts and terminologies
- Introduction to SQL and its development environments
- The HR schema and the tables used in this course
- Oracle Database 11g documentation and additional resources

# Course Objectives

After completing this course, you should be able to:

- Identify the major components of Oracle Database 11g
- Retrieve row and column data from tables with the `SELECT` statement
- Create reports of sorted and restricted data
- Employ SQL functions to generate and retrieve customized data
- Run complex queries to retrieve data from multiple tables
- Run data manipulation language (DML) statements to update data in Oracle Database 11g
- Run data definition language (DDL) statements to create and manage schema objects

# Course Agenda

- Day 1:
  - Introduction
  - Retrieving Data Using the SQL `SELECT` Statement
  - Restricting and Sorting Data
  - Using Single-Row Functions to Customize Output
  - Using Conversion Functions and Conditional Expressions
- Day 2:
  - Reporting Aggregated Data Using the Group Functions
  - Displaying Data from Multiple Tables
  - Using Subqueries to Solve Queries
  - Using the Set Operators

# Course Agenda

- Day 3:
  - Manipulating Data
  - Using DDL Statements to Create and Manage Tables
  - Creating Other Schema Objects



# Appendixes Used in the Course

- Appendix A: Practice Solutions
- Appendix B: Table Descriptions
- Appendix C: Oracle Join Syntax
- Appendix D: Using SQL\*Plus
- Appendix E: Using SQL Developer
- Additional Practices
- Additional Practices Solutions

# Lesson Agenda

- Course objectives, course agenda, and appendixes used in this course
- **Overview of Oracle Database 11g and related products**
- Overview of relational database management concepts and terminologies
- Introduction to SQL and its development environments
- The HR schema and the tables used in this course
- Oracle Database 11g documentation and additional resources

# Oracle Database 11g: Focus Areas



**Infrastructure  
Grids**

**Information  
Management**

**Application  
Development**

# Oracle Database 11g



**Manageability**  
**High availability**  
**Performance**  
**Security**  
**Information integration**

# Oracle Database 11g



# Oracle Fusion Middleware

Portfolio of leading, standards-based, and customer-proven software products that spans a range of tools and services from J2EE and developer tools, through integration services, business intelligence, collaboration, and content management



# Oracle Enterprise Manager Grid Control 10g

- Efficient Oracle Fusion Middleware management
- Simplifying application and infrastructure life cycle management
- Improved database administration and application management capabilities



# Oracle BI Publisher

- Provides a central architecture for authoring, managing, and delivering information in secure and multiple formats
- Reduces complexity and time to develop, test, and deploy all kinds of reports
  - Financial Reports, Invoices, Sales or Purchase orders, XML, and EDI/EFT(eText documents)
- Enables flexible customizations
  - For example, a Microsoft Word document report can be generated in multiple formats such as PDF, HTML, Excel, RTF, and so on.





# Lesson Agenda

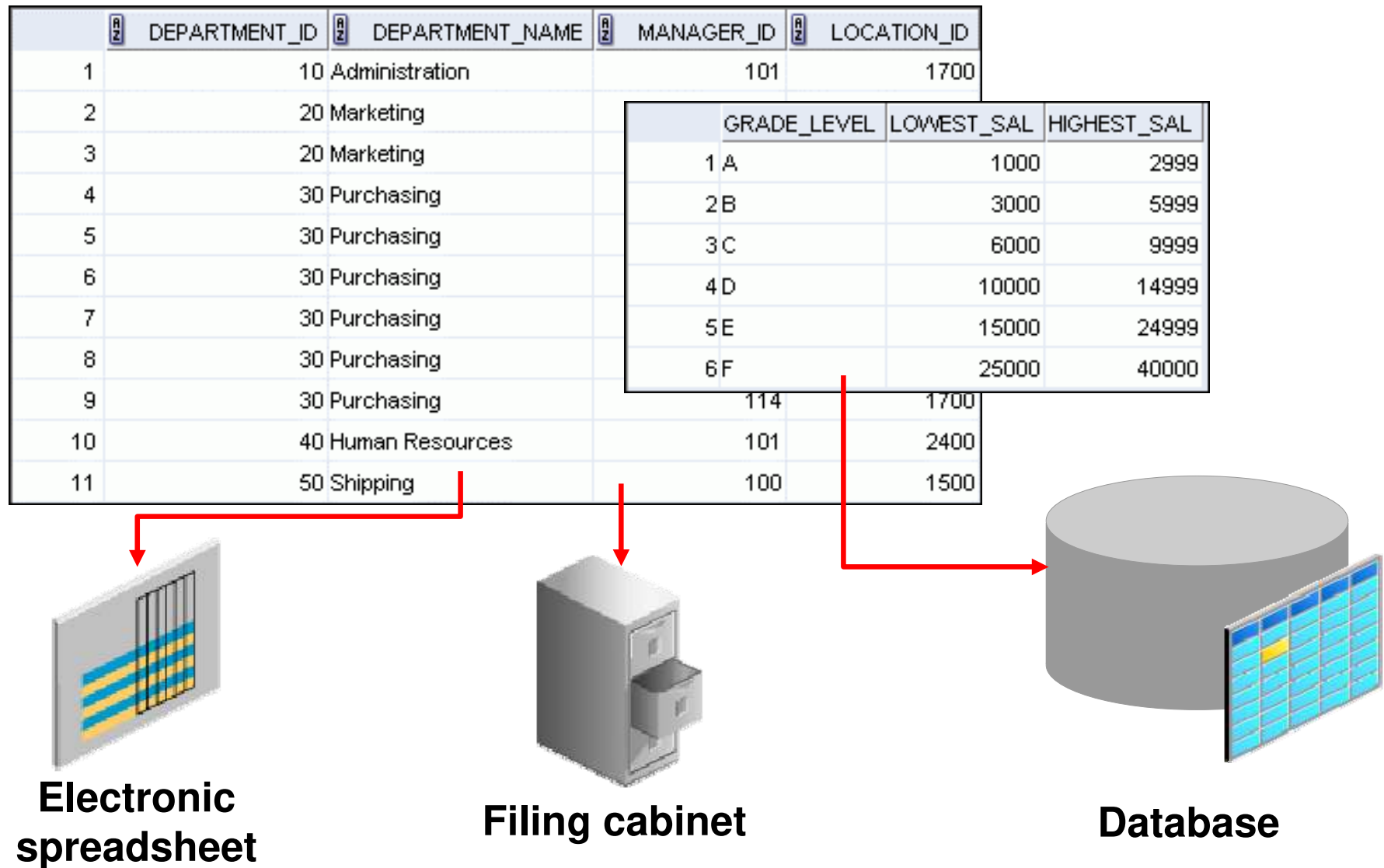
- Course objectives, course agenda, and appendixes used in this course
- Overview of Oracle Database 11g and related products
- Overview of relational database management concepts and terminologies
- Introduction to SQL and its development environments
- The HR schema and the tables used in this course
- Oracle Database 11g documentation and additional resources

# Relational and Object Relational Database Management Systems

- Relational model and object relational model
- User-defined data types and objects
- Fully compatible with relational database
- Supports multimedia and large objects
- High-quality database server features



# Data Storage on Different Media

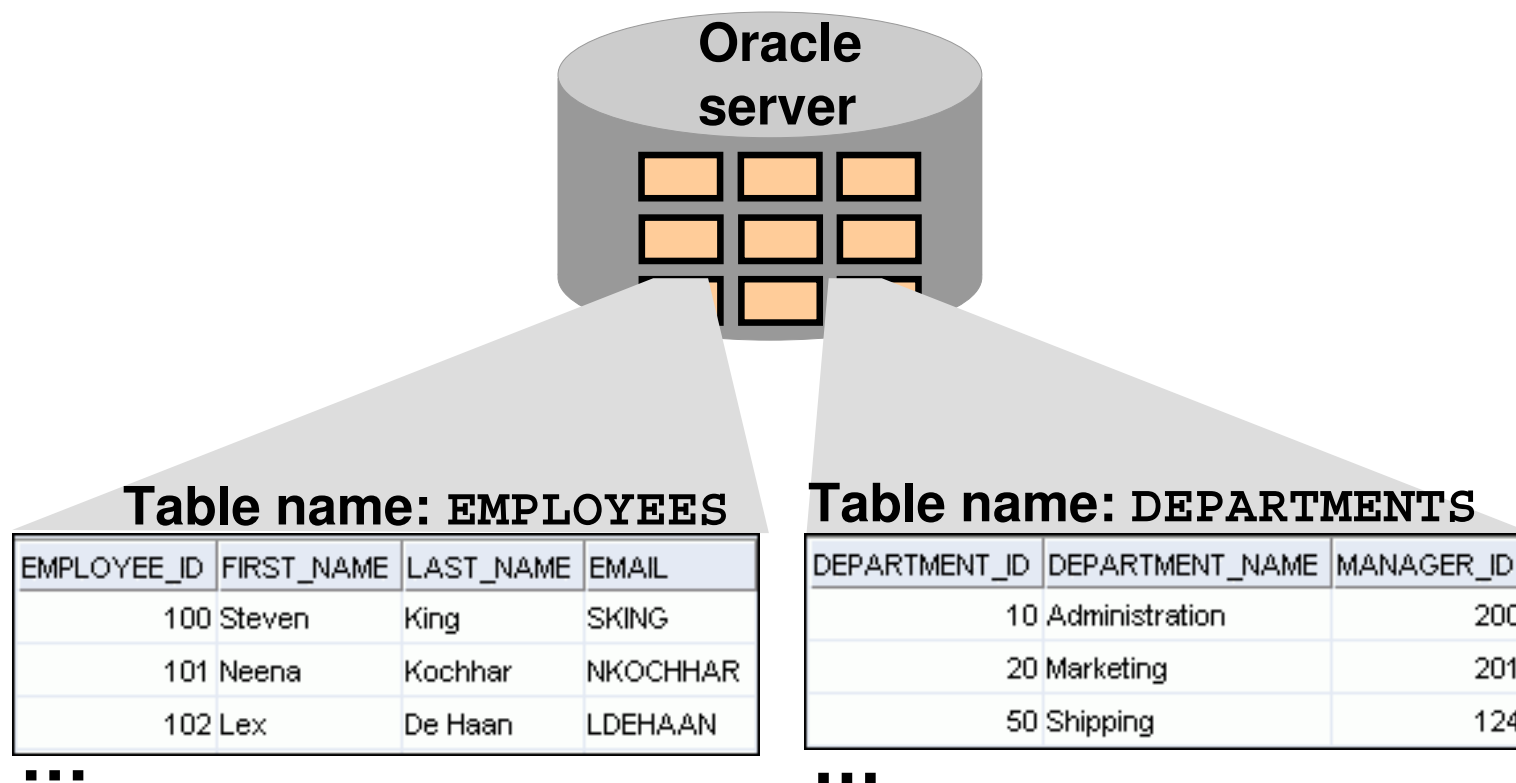


# Relational Database Concept

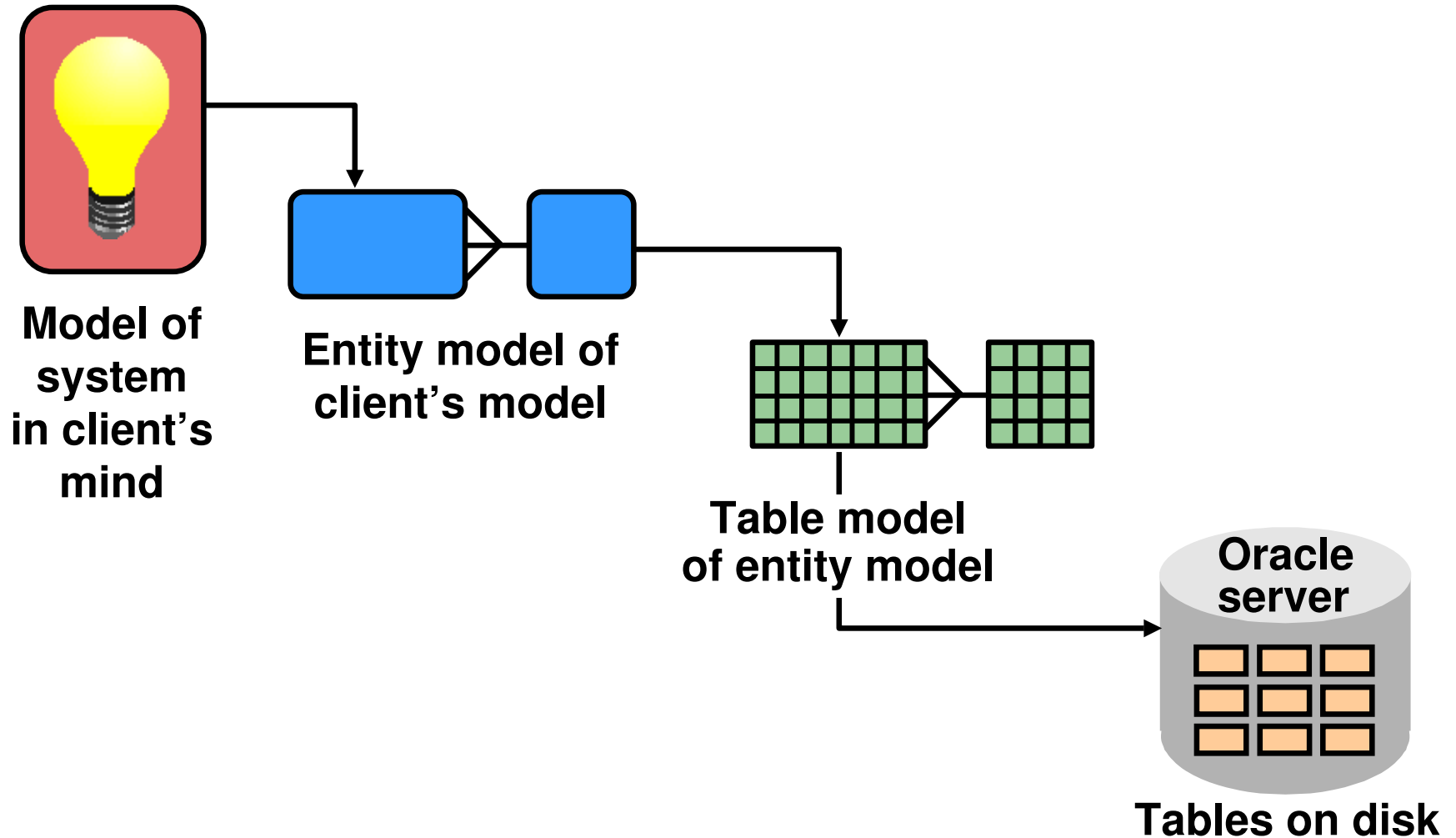
- Dr. E. F. Codd proposed the relational model for database systems in 1970.
- It is the basis for the relational database management system (RDBMS).
- The relational model consists of the following:
  - Collection of objects or relations
  - Set of operators to act on the relations
  - Data integrity for accuracy and consistency

# Definition of a Relational Database

A relational database is a collection of relations or two-dimensional tables.

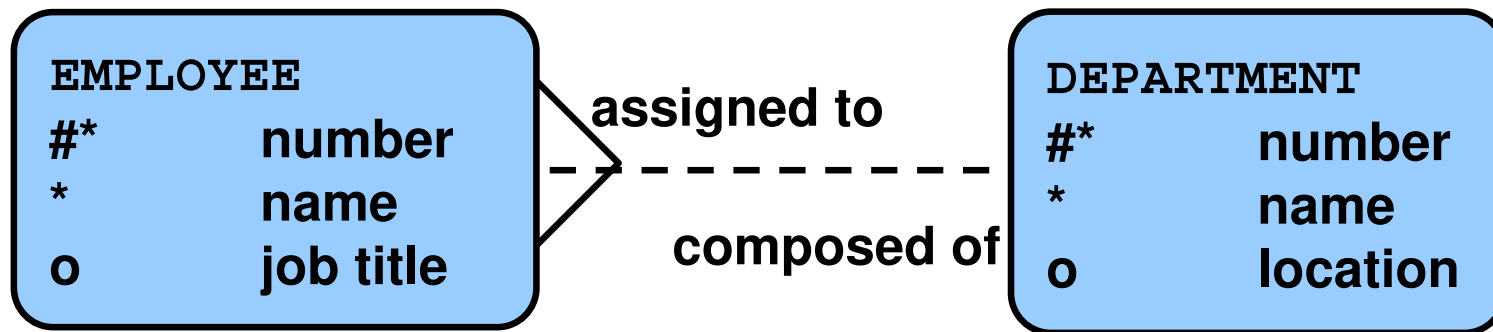


# Data Models



# Entity Relationship Model

- Create an entity relationship diagram from business specifications or narratives:



- Scenario:
  - "... Assign one or more employees to a department ..."
  - "... Some departments do not yet have assigned employees ..."

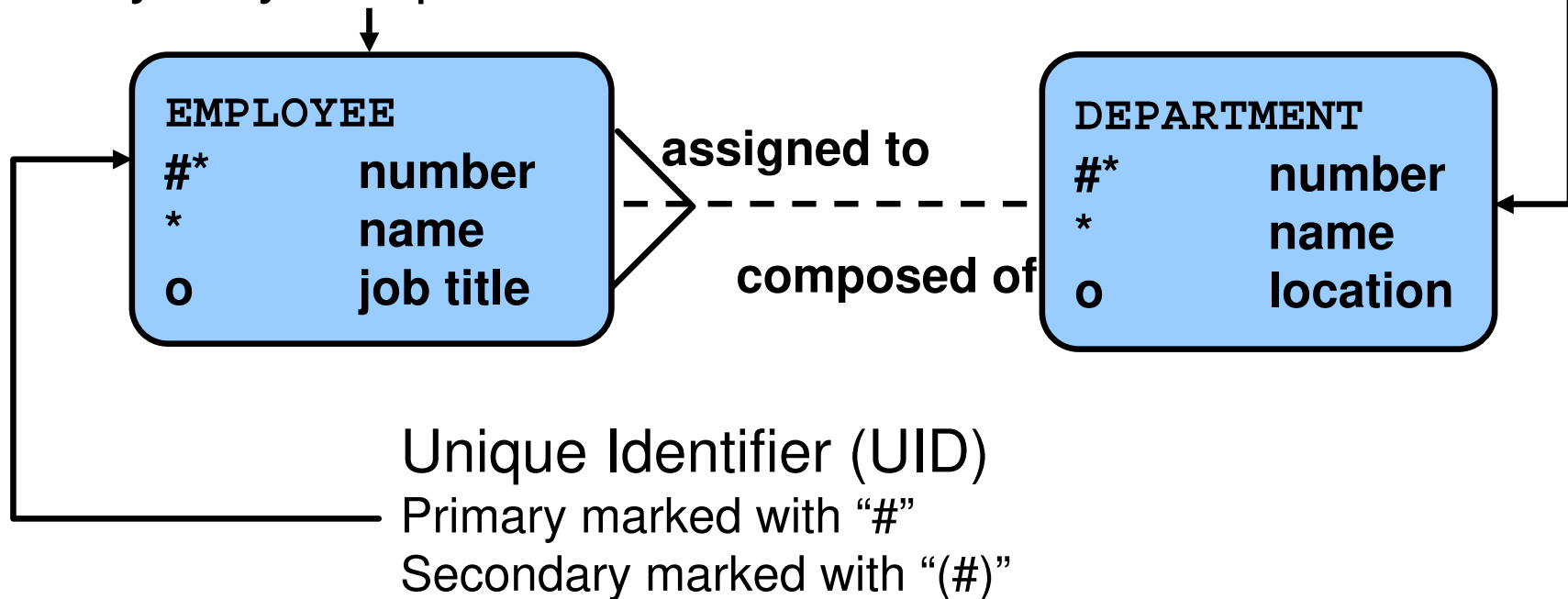
# Entity Relationship Modeling Conventions

## Entity:

- Singular, unique name
- Uppercase
- Soft box
- Synonym in parentheses

## Attribute:

- Singular name
- Lowercase
- Mandatory marked with “\*”
- Optional marked with “o”





# Relating Multiple Tables

- Each row of data in a table is uniquely identified by a primary key.
- You can logically relate data from multiple tables using foreign keys.

**Table name: EMPLOYEES**

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID
100	Steven	King	90
101	Neena	Kochhar	90
102	Lex	De Haan	90
103	Alexander	Hunold	60
104	Bruce	Ernst	60
107	Diana	Lorentz	60

...

**Primary key**

**Foreign key**

**Primary key**

**Table name: DEPARTMENTS**

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

# Relational Database Terminology

2	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	COMMISSION_PCT	DEPARTMENT_ID	4
	100	Steven	King	24000	(null)	90	
	101	Neena	Kochhar	17000	(null)	90	
	102	Lex	De Haan	17000	(null)	90	
	103	Alexander	Hunold	9000	(null)	60	
	104	Bruce	Ernst	6000	(null)	60	5
	107	Diana	Lorentz	4200	(null)	60	
	124	Kevin	Mourgos	5800	(null)	50	
	141	Trenna	Rajs	3500	(null)	50	
	142	Curtis	Davies	3100	(null)	50	
	143	Randall	Matos	2600	(null)	50	
	144	Peter	Vargas	2500	(null)	50	
	149	Eleni	Zlotkey	10500	0.2	80	
	174	Ellen	Abel	11000	0.3	80	
	176	Jonathon	Taylor	8600	0.2	80	
	178	Kimberely	Grant	7000	0.15	(null)	
	200	Jennifer	Whalen	4400	(null)	10	
1	201	Michael	Hartstein	13000	(null)	20	
	202	Pat	Fay	6000	(null)	20	
	205	Shelley	Higgins	12000	(null)	110	
	206	William	Gietz	8300	(null)	110	

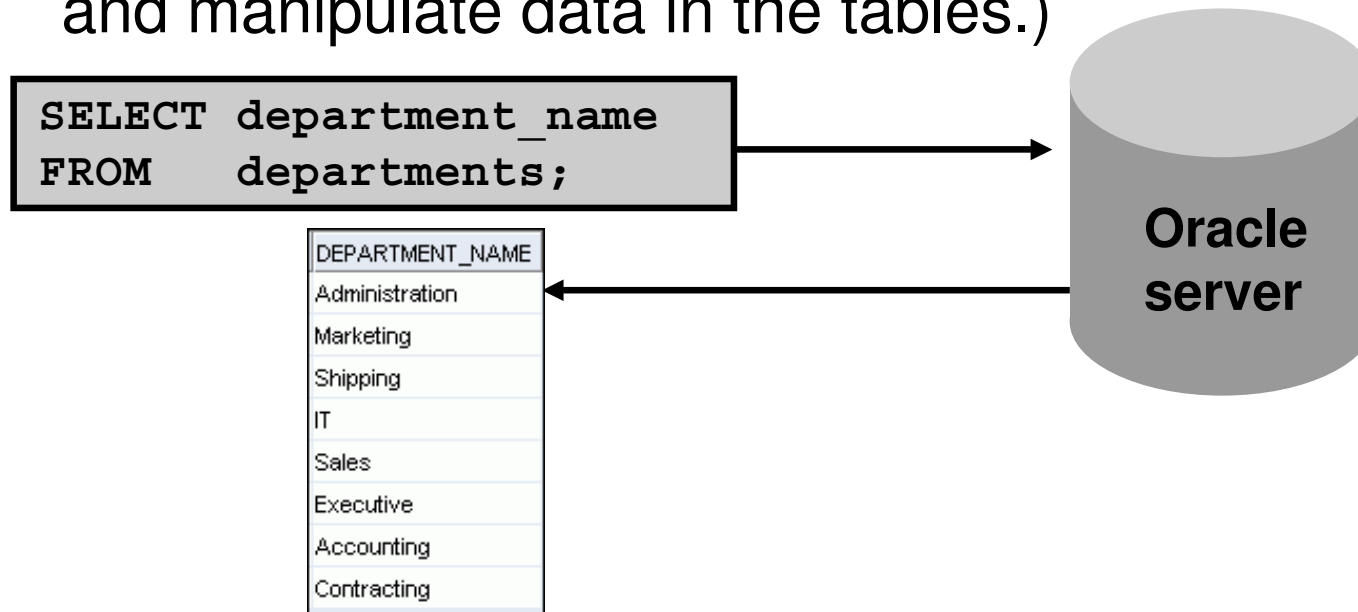
# Lesson Agenda

- Course objectives, course agenda, and appendixes used in this course
- Overview of Oracle Database 11g and related products
- Overview of relational database management concepts and terminologies
- **Introduction to SQL and its development environments**
- The HR schema and the tables used in this course
- Oracle Database 11g documentation and additional resources

# Using SQL to Query Your Database

Structured query language (SQL) is:

- The ANSI standard language for operating relational databases
- Efficient, easy to learn, and use
- Functionally complete (With SQL, you can define, retrieve, and manipulate data in the tables.)



# SQL Statements

SELECT  
INSERT  
UPDATE  
DELETE  
MERGE

Data manipulation language (DML)

CREATE  
ALTER  
DROP  
RENAME  
TRUNCATE  
COMMENT

Data definition language (DDL)

GRANT  
REVOKE

Data control language (DCL)

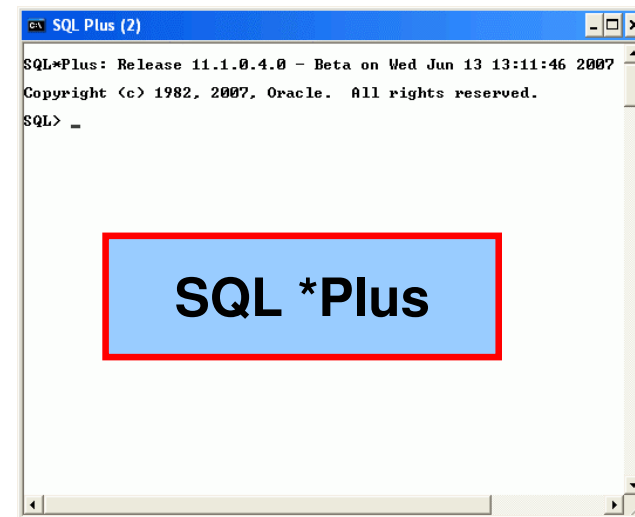
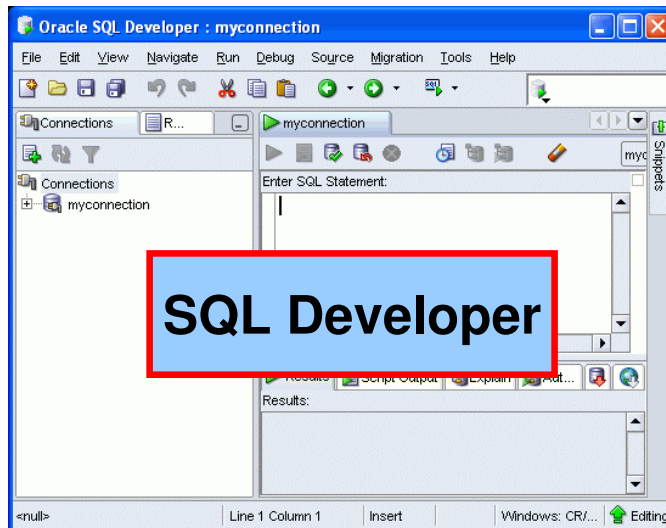
COMMIT  
ROLLBACK  
SAVEPOINT

Transaction control

# Development Environments for SQL

There are two development environments for this course:

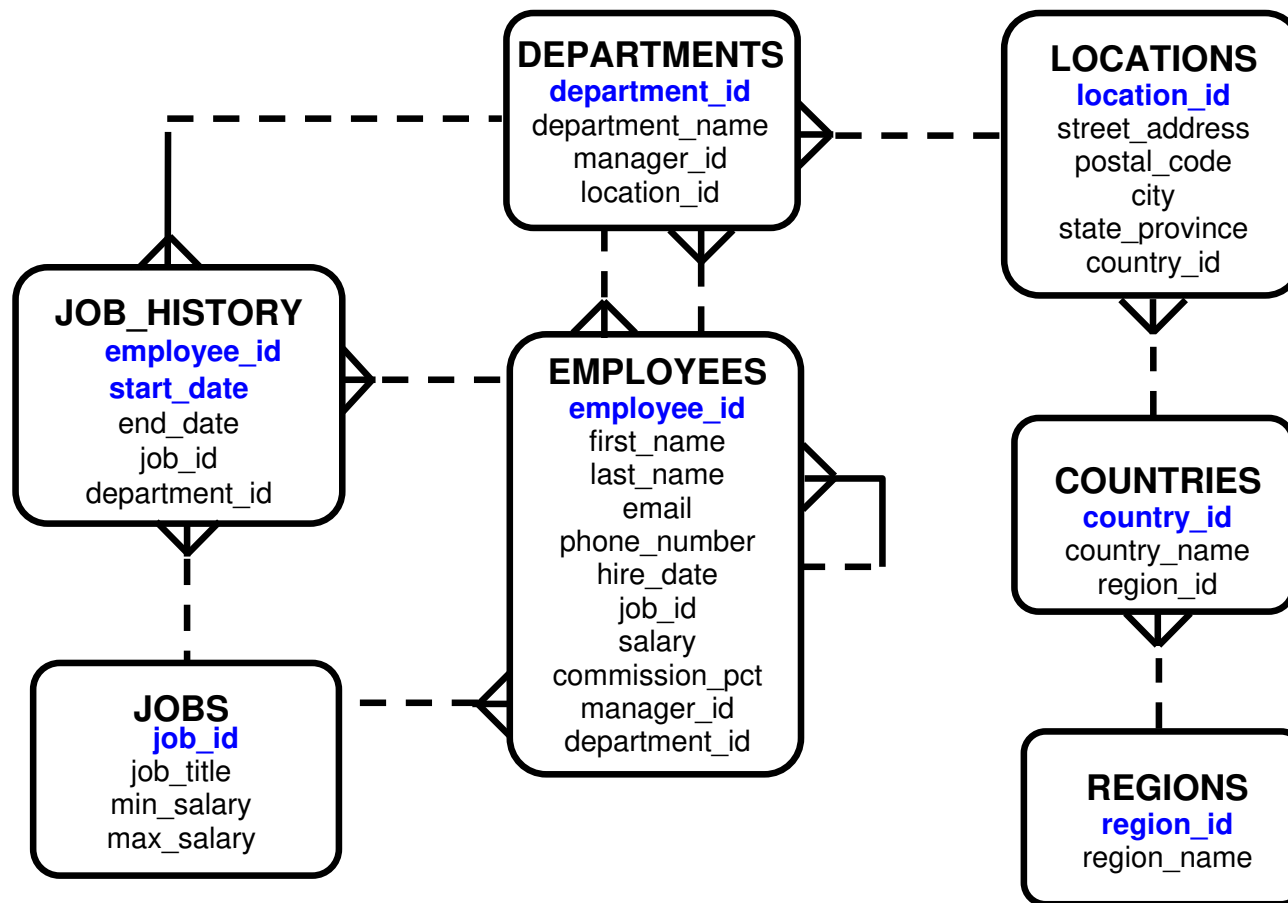
- Primary tool is Oracle SQL Developer
- SQL\*Plus command line interface may also be used



# Lesson Agenda

- Course objectives, course agenda, and appendixes used in this course
- Overview of Oracle Database 11g and related products
- Overview of relational database management concepts and terminologies
- Introduction to SQL and its development environments
- **The HR schema and the tables used in this course**
- Oracle Database 11g documentation and additional resources

# The Human Resources (HR) Schema





# Tables Used in the Course

## EMPLOYEES

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
					50	RMATOS	650.121.2874	15-MAR-98

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

GRADE_LEVEL	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000

## DEPARTMENTS

## JOB\_GRADES

# Lesson Agenda

- Course objectives, course agenda, and appendixes used in this course
- Overview of Oracle Database 11g and related products
- Overview of relational database management concepts and terminologies
- Introduction to SQL and its development environments
- The HR schema and the tables used in this course
- Oracle Database 11g documentation and additional resources

# Oracle Database 11g Documentation

- *Oracle Database New Features Guide 11g, Release 1 (11.1)*
- *Oracle Database Reference 11g, Release 1 (11.1)*
- *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*
- *Oracle Database Concepts 11g, Release 1 (11.1)*
- *Oracle Database SQL Developer User's Guide, Release 1.2*

# Additional Resources

For additional information about the Oracle Database 11g, refer to the following:

- *Oracle Database 11g: New Features eStudies*
- *Oracle by Example series (OBE): Oracle Database 11g*
  - [http://www.oracle.com/technology/obe/11gr1\\_db/index.htm](http://www.oracle.com/technology/obe/11gr1_db/index.htm)

# Summary

In this lesson, you should have learned that:

- Oracle Database 11g extends:
  - The benefits of infrastructure grids
  - The existing information management capabilities
  - The capabilities to use the major application development environments such as PL/SQL, Java/JDBC, .NET, XML, and so on
- The database is based on ORDBMS
- Relational databases are composed of relations, managed by relational operations, and governed by data integrity constraints
- With the Oracle server, you can store and manage information by using SQL

# Practice I: Overview

This practice covers the following topics:

- Running the Oracle SQL Developer demo
- Starting Oracle SQL Developer, creating a new database connection, and browsing the HR tables



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

# Objectives

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

- List the capabilities of SQL `SELECT` statements
- Execute a basic `SELECT` statement



# Lesson Agenda

- Basic `SELECT` statement
- Arithmetic expressions and `NULL` values in the `SELECT` statement
- Column aliases
- Use of concatenation operator, literal character strings, alternative quote operator, and the `DISTINCT` keyword
- `DESCRIBE` command

# Capabilities of SQL `SELECT` Statements

## Projection

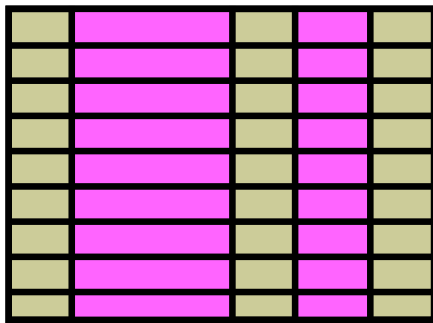



Table 1

## Selection

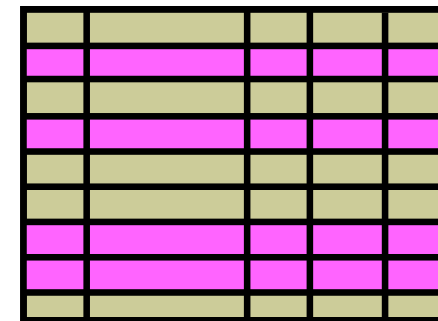



Table 1

## Join

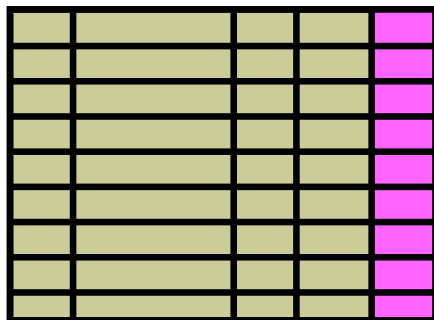



Table 1

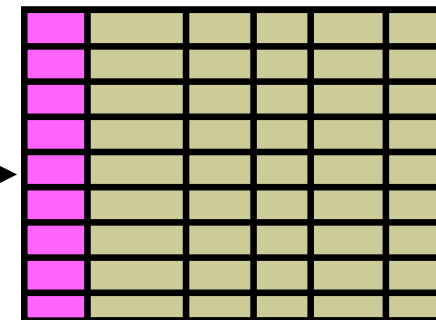



Table 2



# Basic SELECT Statement

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

- SELECT identifies the columns to be displayed.
- FROM identifies the table containing those columns.

# Selecting All Columns

```
SELECT *  
FROM 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

# Selecting Specific Columns

```
SELECT department_id, location_id  
FROM departments;
```

	DEPARTMENT_ID	LOCATION_ID
1	10	1700
2	20	1800
3	50	1500
4	60	1400
5	80	2500
6	90	1700
7	110	1700
8	190	1700

# Writing SQL Statements

- SQL statements are not case-sensitive.
- SQL statements can be entered on one or more lines.
- Keywords cannot be abbreviated or split across lines.
- Clauses are usually placed on separate lines.
- Indents are used to enhance readability.
- In SQL Developer, SQL statements can optionally be terminated by a semicolon (;). Semicolons are required when you execute multiple SQL statements.
- In SQL\*Plus, you are required to end each SQL statement with a semicolon (;).

# Column Heading Defaults

- SQL Developer:
  - Default heading alignment: Left-aligned
  - Default heading display: Uppercase
- SQL\*Plus:
  - Character and Date column headings are left-aligned.
  - Number column headings are right-aligned.
  - Default heading display: Uppercase

# Lesson Agenda

- Basic `SELECT` statement
- Arithmetic expressions and `NULL` values in the `SELECT` statement
- Column Aliases
- Use of concatenation operator, literal character strings, alternative quote operator, and the `DISTINCT` keyword
- `DESCRIBE` command






# Arithmetic Expressions

Create expressions with number and date data by using arithmetic operators.

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide

# Using Arithmetic Operators

```
SELECT last_name, salary, salary + 300  
FROM employees;
```

	 LAST_NAME	 SALARY	 SALARY+300
1	King	24000	24300
2	Kochhar	17000	17300
3	De Haan	17000	17300
4	Hunold	9000	9300
5	Ernst	6000	6300
6	Lorentz	4200	4500
7	Mourgos	5800	6100
8	Rajs	3500	3800
9	Davies	3100	3400
10	Matos	2600	2900

...

# Operator Precedence

```
SELECT last_name, salary, 12*salary+100
FROM employees;
```

1

	R2	LAST_NAME	R2	SALARY	R2	12*SALARY+100
1		King		24000		288100
2		Kochhar		17000		204100
3		De Haan		17000		204100

...

```
SELECT last_name, salary, 12*(salary+100)
FROM employees;
```

2

	R2	LAST_NAME	R2	SALARY	R2	12*(SALARY+100)
1		King		24000		289200
2		Kochhar		17000		205200
3		De Haan		17000		205200

...

# Defining a Null Value

- Null is a value that is unavailable, unassigned, unknown, or inapplicable.
- Null is not the same as zero or a blank space.

```
SELECT last_name, job_id, salary, commission_pct  
FROM   employees;
```

	LAST_NAME	JOB_ID	SALARY	COMMISSION_PCT
1	King	AD_PRES	24000	(null)
2	Kochhar	AD_VP	17000	(null)

...

12	Zlotkey	SA_MAN	10500	0.2
13	Abel	SA_REP	11000	0.3
14	Taylor	SA_REP	8600	0.2

...

19	Higgins	AC_MGR	12000	(null)
20	Gietz	AC_ACCOUNT	8300	(null)

# Null Values in Arithmetic Expressions

Arithmetic expressions containing a null value evaluate to null.

```
SELECT last_name, 12*salary*commission_pct  
FROM employees;
```

	A2	LAST_NAME	A2	12*SALARY*COMMISSION_PCT
1		King		(null)
2		Kochhar		(null)
...				
12		Zlotkey		25200
13		Abel		39600
14		Taylor		20640
...				
19		Higgins		(null)
20		Gietz		(null)

# Lesson Agenda

- Basic `SELECT` statement
- Arithmetic expressions and `NULL` values in the `SELECT` statement
- **Column aliases**
- Use of concatenation operator, literal character strings, alternative quote operator, and the `DISTINCT` keyword
- `DESCRIBE` command

# Defining a Column Alias

A column alias:

- Renames a column heading
- Is useful with calculations
- Immediately follows the column name (There can also be the optional `AS` keyword between the column name and alias.)
- Requires double quotation marks if it contains spaces or special characters, or if it is case-sensitive

# Using Column Aliases

```
SELECT last_name AS name, commission_pct comm  
FROM employees;
```

	NAME	COMM
1	King	(null)
2	Kochhar	(null)
3	De Haan	(null)

...

```
SELECT last_name "Name", salary*12 "Annual Salary"  
FROM employees;
```

	Name	Annual Salary
1	King	288000
2	Kochhar	204000
3	De Haan	204000

...



# Lesson Agenda

- Basic `SELECT` Statement
- Arithmetic Expressions and `NULL` values in `SELECT` statement
- Column Aliases
- Use of concatenation operator, literal character strings, alternative quote operator, and the `DISTINCT` keyword
- `DESCRIBE` command

# Concatenation Operator

A concatenation operator:

- Links columns or character strings to other columns
- Is represented by two vertical bars (||)
- Creates a resultant column that is a character expression

```
SELECT    last_name || job_id AS "Employees"  
FROM      employees;
```

	A Z	Employees
1		AbelSA_REP
2		DaviesST_CLERK
3		De HaanAD_VP
4		ErnstIT_PROG
5		FayMK_REP

...

# Literal Character Strings

- A literal is a character, a number, or a date that is included in the `SELECT` statement.
- Date and character literal values must be enclosed within single quotation marks.
- Each character string is output once for each row returned.

# Using Literal Character Strings

```
SELECT last_name || ' is a ' || job_id  
       AS "Employee Details"  
FROM   employees;
```

	Employee Details
1	Abel is a SA_REP
2	Davies is a ST_CLERK
3	De Haan is a AD_VP
4	Ernst is a IT_PROG
5	Fay is a MK_REP

...

18	Vargas is a ST_CLERK
19	Whalen is a AD_ASST
20	Zlotkey is a SA_MAN

# Alternative Quote (q) Operator

- Specify your own quotation mark delimiter.
- Select any delimiter.
- Increase readability and usability.

```
SELECT department_name || q' [ Department's Manager Id: ] '  
      || manager_id  
      AS "Department and Manager"  
FROM departments;
```

	Department and Manager
1	Administration Department's Manager Id:200
2	Marketing Department's Manager Id:201
3	Shipping Department's Manager Id:124
4	IT Department's Manager Id:103
5	Sales Department's Manager Id:149
6	Executive Department's Manager Id:100
7	Accounting Department's Manager Id:205
8	Contracting Department's Manager Id:

# Duplicate Rows

The default display of queries is all rows, including duplicate rows.

```
SELECT department_id  
FROM employees;
```

1

	DEPARTMENT_ID
1	90
2	90
3	90
4	60
5	60

...

```
SELECT DISTINCT department_id  
FROM employees;
```

2

	DEPARTMENT_ID
1	(null)
2	90
3	20
...	
4	110

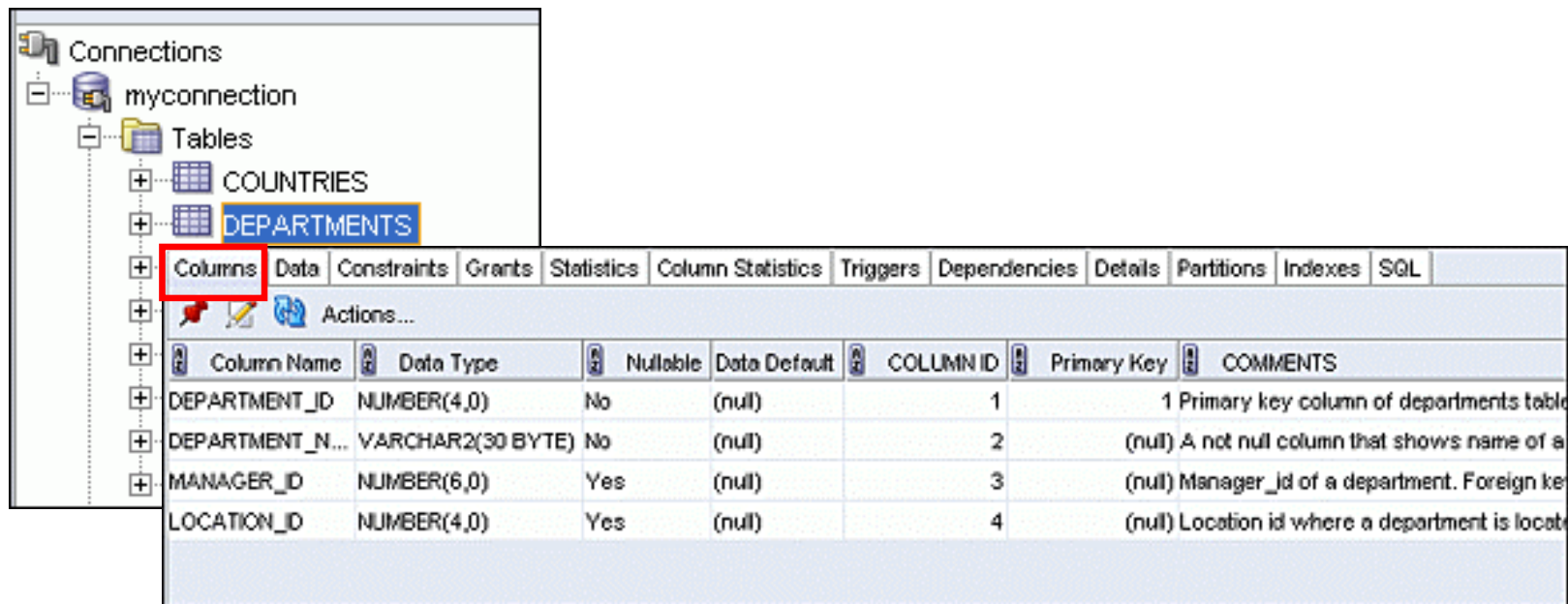
# Lesson Agenda

- Basic `SELECT` statement
- Arithmetic expressions and `NULL` values in the `SELECT` statement
- Column aliases
- Use of concatenation operator, literal character strings, alternative quote operator, and the `DISTINCT` keyword
- **`DESCRIBE` command**

# Displaying the Table Structure

- Use the DESCRIBE command to display the structure of a table.
- Or, select the table in the Connections tree and use the Columns tab to view the table structure.

```
DESC[RIBE] tablename
```



The screenshot shows the Oracle SQL Developer interface. On the left, the 'Connections' tree is expanded, showing 'myconnection' > 'Tables' > 'DEPARTMENTS'. The 'Columns' tab is selected and highlighted with a red box. The main pane displays the table structure for 'DEPARTMENTS'.

Column Name	Data Type	Nullable	Data Default	COLUMN ID	Primary Key	COMMENTS
DEPARTMENT_ID	NUMBER(4,0)	No	(null)	1	1	Primary key column of departments table
DEPARTMENT_N...	VARCHAR2(30 BYTE)	No	(null)	2	(null)	A not null column that shows name of a
MANAGER_ID	NUMBER(6,0)	Yes	(null)	3	(null)	Manager_id of a department. Foreign ke
LOCATION_ID	NUMBER(4,0)	Yes	(null)	4	(null)	Location id where a department is locat



# Using the DESCRIBE Command

```
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)

11 rows selected
```

# Quiz

Identify the SELECT statements that execute successfully.

1.

```
SELECT first_name, last_name, job_id, salary*12
  AS Yearly Sal
FROM    employees;
```

2.

```
SELECT first_name, last_name, job_id, salary*12
  yearly sal
FROM    employees;
```

3.

```
SELECT first_name, last_name, job_id, salary AS
  yearly sal
FROM    employees;
```

4.

```
SELECT first_name+last_name AS name, job_Id,
  salary*12 yearly sal
FROM    employees;
```

# Summary

In this lesson, you should have learned how to:

- Write a `SELECT` statement that:
  - Returns all rows and columns from a table
  - Returns specified columns from a table
  - Uses column aliases to display more descriptive column headings

```
SELECT * | { [DISTINCT] column/expression [alias], ... }  
FROM table;
```

# Practice 1: Overview

This practice covers the following topics:

- Selecting all data from different tables
- Describing the structure of tables
- Performing arithmetic calculations and specifying column names



# **Restricting and Sorting Data**

# Objectives

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





- Limit the rows that are retrieved by a query
- Sort the rows that are retrieved by a query
- Use ampersand substitution to restrict and sort output at run time

# Lesson Agenda

- Limiting rows with:
  - The `WHERE` clause
  - The comparison conditions using `=`, `<=`, `BETWEEN`, `IN`, `LIKE`, and `NULL` conditions
  - Logical conditions using `AND`, `OR`, and `NOT` operators
- Rules of precedence for operators in an expression
- Sorting rows using the `ORDER BY` clause
- Substitution variables
- `DEFINE` and `VERIFY` commands

# Limiting Rows Using a Selection




## EMPLOYEES

	 EMPLOYEE_ID	 LAST_NAME	 JOB_ID	 DEPARTMENT_ID
1	100	King	AD_PRES	90
2	101	Kochhar	AD_VP	90
3	102	De Haan	AD_VP	90
4	103	Hunold	IT_PROG	60
5	104	Ernst	IT_PROG	60
6	107	Lorentz	IT_PROG	60

...

**“retrieve all  
employees in  
department 90”**



	 EMPLOYEE_ID	 LAST_NAME	 JOB_ID	 DEPARTMENT_ID
1	100	King	AD_PRES	90
2	101	Kochhar	AD_VP	90
3	102	De Haan	AD_VP	90



# Limiting the Rows That Are Selected

- Restrict the rows that are returned by using the WHERE clause:

```
SELECT * | { [DISTINCT] column/expression [alias], ... }  
FROM    table  
[WHERE condition(s)];
```

- The WHERE clause follows the FROM clause.

# Using the WHERE Clause

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees
WHERE  department_id = 90 ;
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	100	King	AD_PRES	90
2	101	Kochhar	AD_VP	90
3	102	De Haan	AD_VP	90

# Character Strings and Dates

- Character strings and date values are enclosed with single quotation marks.
- Character values are case-sensitive and date values are format-sensitive.
- The default date display format is DD-MON-RR.

```
SELECT last_name, job_id, department_id
FROM   employees
WHERE  last_name = 'Whalen' ;
```

```
SELECT last_name
FROM   employees
WHERE  hire_date = '17-FEB-96' ;
```

# Comparison Operators

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
BETWEEN ...AND...	Between two values (inclusive)
IN (set)	Match any of a list of values
LIKE	Match a character pattern
IS NULL	Is a null value

# Using Comparison Operators

```
SELECT last_name, salary
FROM   employees
WHERE  salary <= 3000 ;
```

	LAST_NAME	SALARY
1	Matos	2600
2	Vargas	2500

# Range Conditions Using the BETWEEN Operator

Use the BETWEEN operator to display rows based on a range of values:

```
SELECT last_name, salary
FROM   employees
WHERE  salary BETWEEN 2500 AND 3500 ;
```

↑  
**Lower limit**

↑  
**Upper limit**

	LAST_NAME	SALARY
1	Rajs	3500
2	Davies	3100
3	Matos	2600
4	Vargas	2500

# Membership Condition Using the IN Operator

Use the IN operator to test for values in a list:

```
SELECT employee_id, last_name, salary, manager_id
FROM   employees
WHERE  manager_id IN (100, 101, 201) ;
```

	EMPLOYEE_ID	LAST_NAME	SALARY	MANAGER_ID
1	101	Kochhar	17000	100
2	102	De Haan	17000	100
3	124	Mourgos	5800	100
4	149	Zlotkey	10500	100
5	201	Hartstein	13000	100
6	200	Whalen	4400	101
7	205	Higgins	12000	101
8	202	Fay	6000	201

# Pattern Matching Using the LIKE Operator

- Use the LIKE operator to perform wildcard searches of valid search string values.
- Search conditions can contain either literal characters or numbers:
  - % denotes zero or many characters.
  - \_ denotes one character.

```
SELECT    first_name
FROM      employees
WHERE     first_name LIKE 'S%';
```



# Combining Wildcard Characters

- You can combine the two wildcard characters (% , \_) with literal characters for pattern matching:

```
SELECT last_name  
FROM employees  
WHERE last_name LIKE '_o%' ;
```

	LAST_NAME
1	Kochhar
2	Lorentz
3	Mourgos

- You can use the `ESCAPE` identifier to search for the actual % and \_ symbols.

# Using the NULL Conditions

Test for nulls with the IS NULL operator.

```
SELECT last_name, manager_id
FROM   employees
WHERE  manager_id IS NULL ;
```

	LAST_NAME	MANAGER_ID
1	King	(null)

# Defining Conditions Using the Logical Operators

Operator	Meaning
AND	Returns TRUE if <i>both</i> component conditions are true
OR	Returns TRUE if <i>either</i> component condition is true
NOT	Returns TRUE if the condition is false

# Using the AND Operator

AND requires both the component conditions to be true:

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary >= 10000
AND    job_id LIKE '%MAN%' ;
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	149	Zlotkey	SA_MAN	10500
2	201	Hartstein	MK_MAN	13000

# Using the OR Operator

OR requires either component condition to be true:

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary >= 10000
OR     job_id LIKE '%MAN%' ;
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	100	King	AD_PRES	24000
2	101	Kochhar	AD_VP	17000
3	102	De Haan	AD_VP	17000
4	124	Mourgos	ST_MAN	5800
5	149	Zlotkey	SA_MAN	10500
6	174	Abel	SA_REP	11000
7	201	Hartstein	MK_MAN	13000
8	205	Higgins	AC_MGR	12000

# Using the NOT Operator

```
SELECT last_name, job_id
FROM   employees
WHERE  job_id
      NOT IN ('IT_PROG', 'ST_CLERK', 'SA_REP') ;
```

	LAST_NAME	JOB_ID
1	De Haan	AD_VP
2	Fay	MK_REP
3	Gietz	AC_ACCOUNT
4	Hartstein	MK_MAN
5	Higgins	AC_MGR
6	King	AD_PRES
7	Kochhar	AD_VP
8	Mourgos	ST_MAN
9	Whalen	AD_ASST
10	Zlotkey	SA_MAN

# Lesson Agenda

- Limiting rows with:
  - The `WHERE` clause
  - The comparison conditions using `=`, `<=`, `BETWEEN`, `IN`, `LIKE`, and `NULL` operators
  - Logical conditions using `AND`, `OR`, and `NOT` operators
- Rules of precedence for operators in an expression
- Sorting rows using the `ORDER BY` clause
- Substitution variables
- `DEFINE` and `VERIFY` commands

# Rules of Precedence

Operator	Meaning
1	Arithmetic operators
2	Concatenation operator
3	Comparison conditions
4	IS [NOT] NULL, LIKE, [NOT] IN
5	[NOT] BETWEEN
6	Not equal to
7	NOT logical condition
8	AND logical condition
9	OR logical condition

**You can use parentheses to override rules of precedence.**



# Rules of Precedence

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  job_id = 'SA_REP'
OR     job_id = 'AD_PRES'
AND    salary > 15000;
```

1

	LAST_NAME	JOB_ID	SALARY
1	King	AD_PRES	24000
2	Abel	SA_REP	11000
3	Taylor	SA_REP	8600
4	Grant	SA_REP	7000

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  (job_id = 'SA_REP'
OR     job_id = 'AD_PRES')
AND    salary > 15000;
```

2

	LAST_NAME	JOB_ID	SALARY
1	King	AD_PRES	24000




# Lesson Agenda

- Limiting rows with:
  - The `WHERE` clause
  - The comparison conditions using `=`, `<=`, `BETWEEN`, `IN`, `LIKE`, and `NULL` operators
  - Logical conditions using `AND`, `OR`, and `NOT` operators
- Rules of precedence for operators in an expression
- **Sorting rows using the `ORDER BY` clause**
- Substitution variables
- `DEFINE` and `VERIFY` commands

# Using the ORDER BY Clause

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

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


	 LAST_NAME	 JOB_ID	 DEPARTMENT_ID	HIRE_DATE
1	King	AD_PRES	90	17-JUN-87
2	Whalen	AD_ASST	10	17-SEP-87
3	Kochhar	AD_VP	90	21-SEP-89
4	Hunold	IT_PROG	60	03-JAN-90
5	Ernst	IT_PROG	60	21-MAY-91
6	De Haan	AD_VP	90	13-JAN-93

...

# Sorting


- Sorting in descending order:

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



- Sorting by column alias:


```
SELECT employee_id, last_name, salary*12 annsal  
FROM    employees  
ORDER BY annsal ;
```



# Sorting


- Sorting by using the column's numeric position:

```
SELECT last_name, job_id, department_id, hire_date
FROM employees
ORDER BY 3;
```



- Sorting by multiple columns:

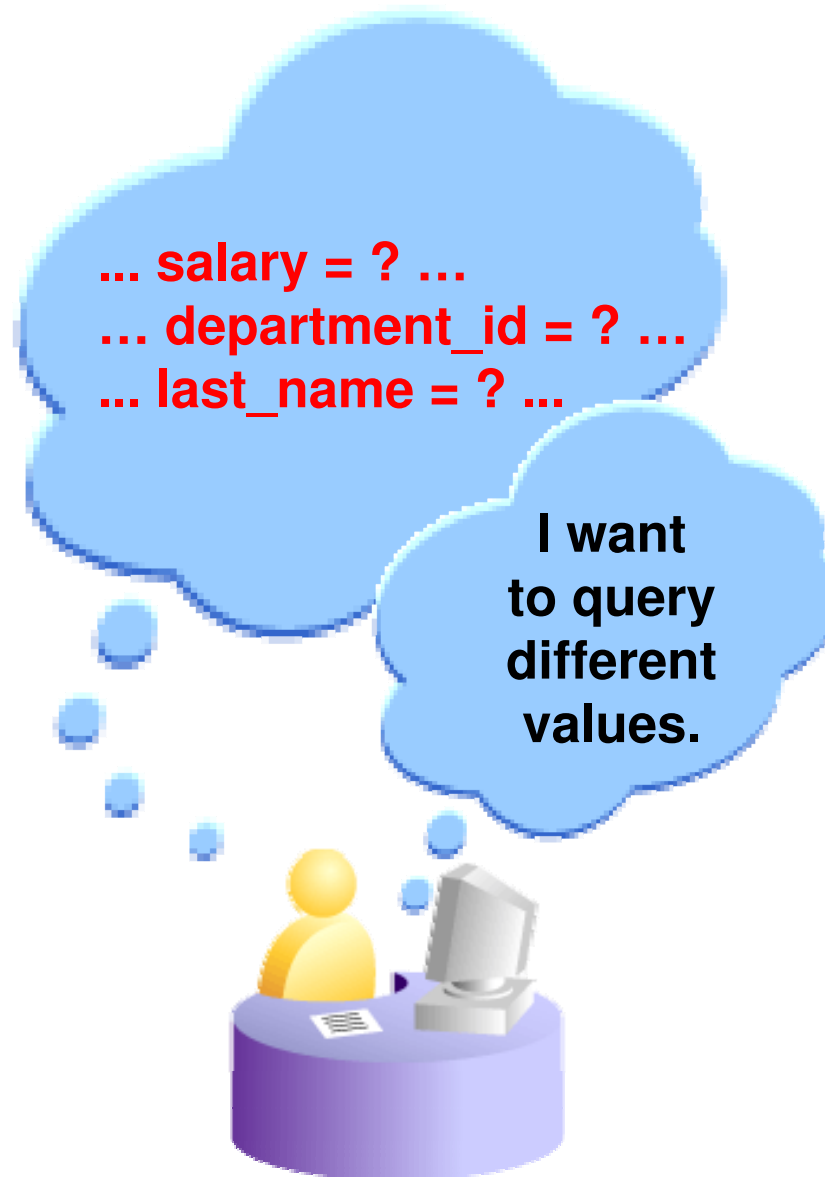
```
SELECT last_name, department_id, salary
FROM employees
ORDER BY department_id, salary DESC;
```



# Lesson Agenda

- Limiting rows with:
  - The `WHERE` clause
  - The comparison conditions using `=`, `<=`, `BETWEEN`, `IN`, `LIKE`, and `NULL` operators
  - Logical conditions using `AND`, `OR`, and `NOT` operators
- Rules of precedence for operators in an expression
- Sorting rows using the `ORDER BY` clause
- **Substitution variables**
- `DEFINE` and `VERIFY` commands

# Substitution Variables



# Substitution Variables

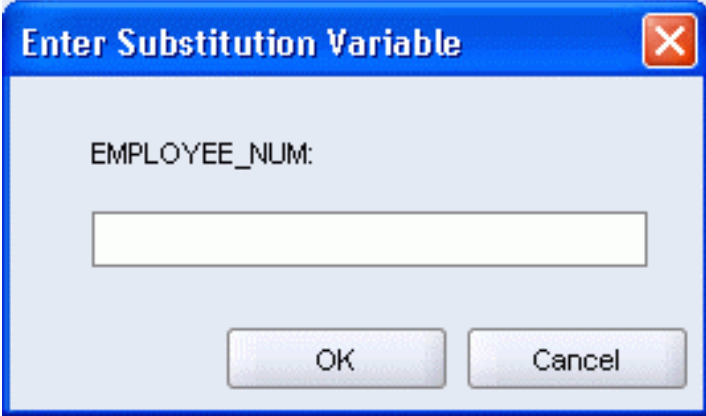
- Use substitution variables to:
  - Temporarily store values with single-ampersand (&) and double-ampersand (&&) substitution
- Use substitution variables to supplement the following:
  - WHERE conditions
  - ORDER BY clauses
  - Column expressions
  - Table names
  - Entire SELECT statements



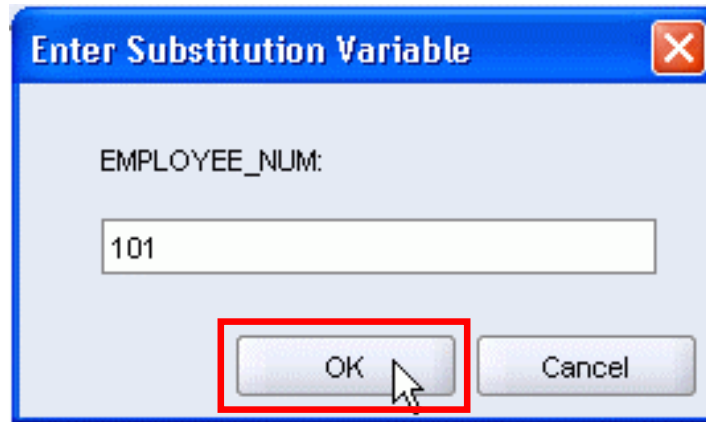
# Using the Single-Ampersand Substitution Variable

Use a variable prefixed with an ampersand (&) to prompt the user for a value:

```
SELECT employee_id, last_name, salary, department_id
FROM   employees
WHERE  employee_id = &employee_num ;
```



# Using the Single-Ampersand Substitution Variable



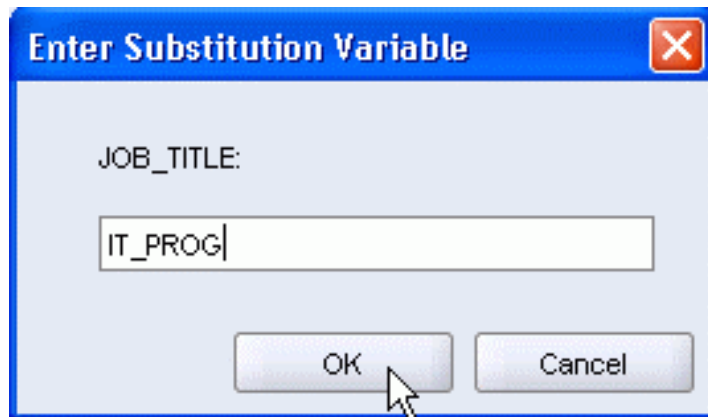
A dialog box titled "Enter Substitution Variable" with a close button (X) in the top right corner. The dialog contains a label "EMPLOYEE\_NUM:" followed by a text input field containing the value "101". Below the input field are two buttons: "OK" and "Cancel". The "OK" button is highlighted with a red rectangular border, and a mouse cursor is pointing at it.

	 EMPLOYEE_ID	 LAST_NAME	 SALARY	 DEPARTMENT_ID
1	101	Kochhar	17000	90

# Character and Date Values with Substitution Variables

Use single quotation marks for date and character values:

```
SELECT last_name, department_id, salary*12
FROM   employees
WHERE  job_id = '&job_title' ;
```

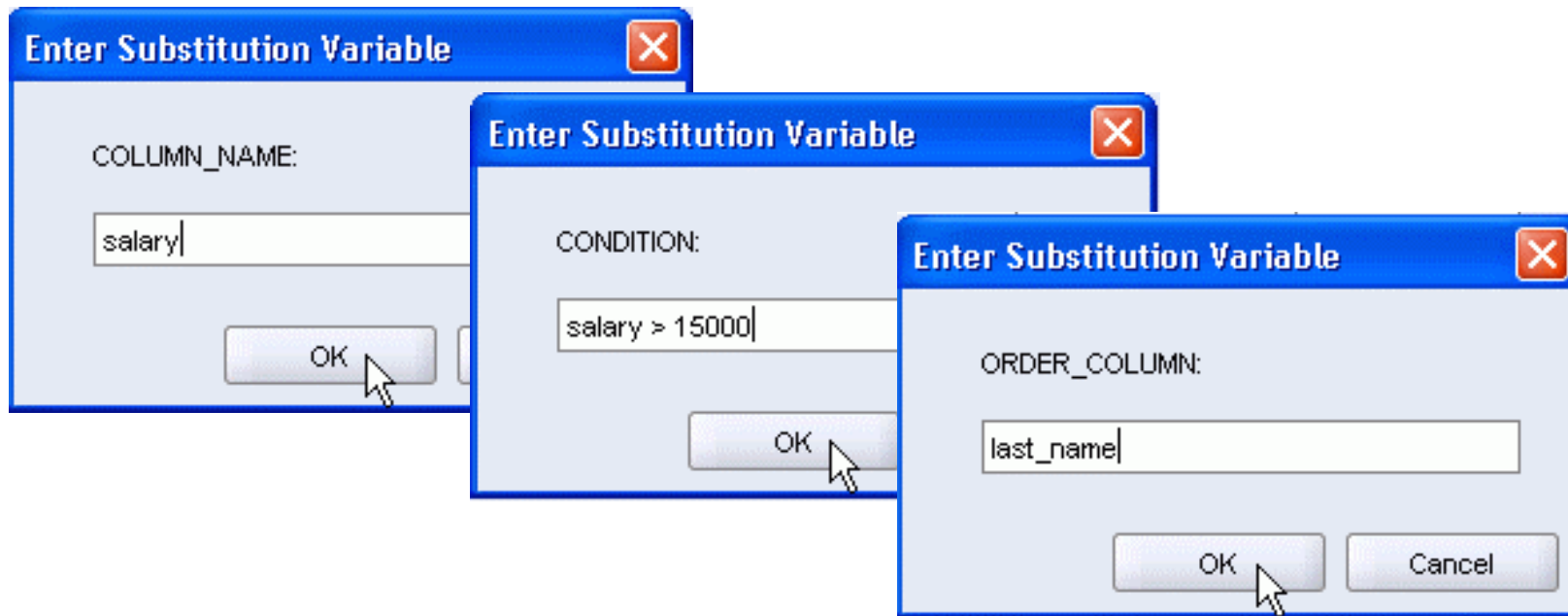


A dialog box titled "Enter Substitution Variable" with a close button (X) in the top right corner. The label "JOB\_TITLE:" is positioned above a text input field. The input field contains the text "IT\_PROG". Below the input field are two buttons: "OK" and "Cancel". A mouse cursor is pointing at the "OK" button.

	LAST_NAME	DEPARTMENT_ID	SALARY*12
1	Hunold	60	108000
2	Ernst	60	72000
3	Lorentz	60	50400

# Specifying Column Names, Expressions, and Text

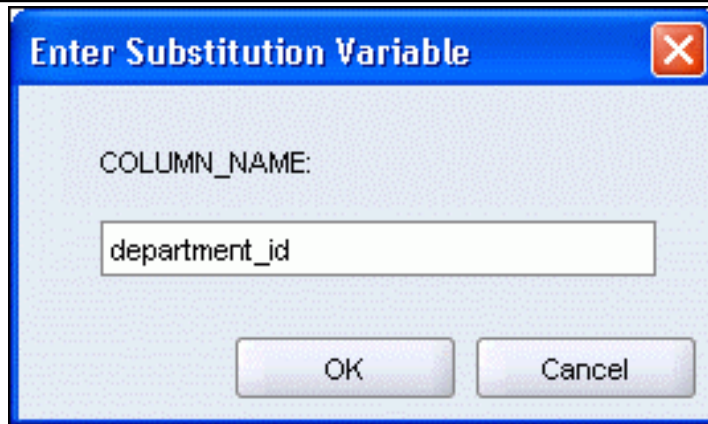
```
SELECT employee_id, last_name, job_id, &column_name  
FROM employees  
WHERE &condition  
ORDER BY &order_column ;
```



# Using the Double-Ampersand Substitution Variable

Use double ampersand (&&) if you want to reuse the variable value without prompting the user each time:

```
SELECT  employee_id, last_name, job_id, &&column_name
FROM    employees
ORDER BY &column_name ;
```



A dialog box titled "Enter Substitution Variable" with a close button (X) in the top right corner. It contains a label "COLUMN\_NAME:" above a text input field. The input field contains the text "department\_id". Below the input field are two buttons: "OK" and "Cancel".

	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	200	Whalen	AD_ASST	10
2	201	Hartstein	MK_MAN	20
3	202	Fay	MK_REP	20

...


# Lesson Agenda

- Limiting rows with:
  - The `WHERE` clause
  - The comparison conditions using `=`, `<=`, `BETWEEN`, `IN`, `LIKE`, and `NULL` operators
  - Logical conditions using `AND`, `OR`, and `NOT` operators
- Rules of precedence for operators in an expression
- Sorting rows using the `ORDER BY` clause
- Substitution variables
- **`DEFINE` and `VERIFY` commands**

# Using the DEFINE Command

- Use the DEFINE command to create and assign a value to a variable.
- Use the UNDEFINE command to remove a variable.

```
DEFINE employee_num = 200  
  
SELECT employee_id, last_name, salary, department_id  
FROM employees  
WHERE employee_id = &employee_num;  
  
UNDEFINE employee_num
```

A red arrow points from the value '200' in the DEFINE statement to the '&employee\_num' placeholder in the WHERE clause of the SQL query, illustrating how the variable's value is substituted.

# Using the VERIFY Command

Use the VERIFY command to toggle the display of the substitution variable, both before and after SQL Developer replaces substitution variables with values:

```
SET VERIFY ON
```

```
SELECT employee_id, last_name, salary  
FROM   employees  
WHERE  employee_id = &employee_num;
```

Enter Substitution Variable

EMPLOYEE\_NUM:

200

OK Cancel

Results Script Output Explain Autotrace DBMS Output

```
SELECT employee_id, last_name, salary  
FROM   employees  
WHERE  employee_id = 200
```

EMPLOYEE_ID	LAST_NAME	SALARY
200	Whalen	4400

1 rows selected



# Quiz

Which of the following are valid operators for the WHERE clause?

1. >=

2. IS NULL

3. !=

4. IS LIKE

5. IN BETWEEN

6. <>

# Summary

In this lesson, you should have learned how to:

- Use the `WHERE` clause to restrict rows of output:
  - Use the comparison conditions
  - Use the `BETWEEN`, `IN`, `LIKE`, and `NULL` operators
  - Apply the logical `AND`, `OR`, and `NOT` operators
- Use the `ORDER BY` clause to sort rows of output:

```
SELECT  * | { [DISTINCT] column/expression [alias], ... }  
FROM    table  
[WHERE  condition(s)]  
[ORDER BY {column, expr, alias} [ASC|DESC]] ;
```

- Use ampersand substitution to restrict and sort output at run time

## Practice 2: Overview

This practice covers the following topics:

- Selecting data and changing the order of the rows that are displayed
- Restricting rows by using the `WHERE` clause
- Sorting rows by using the `ORDER BY` clause
- Using substitution variables to add flexibility to your SQL `SELECT` statements



# Using Single-Row Functions to Customize Output

# Objectives

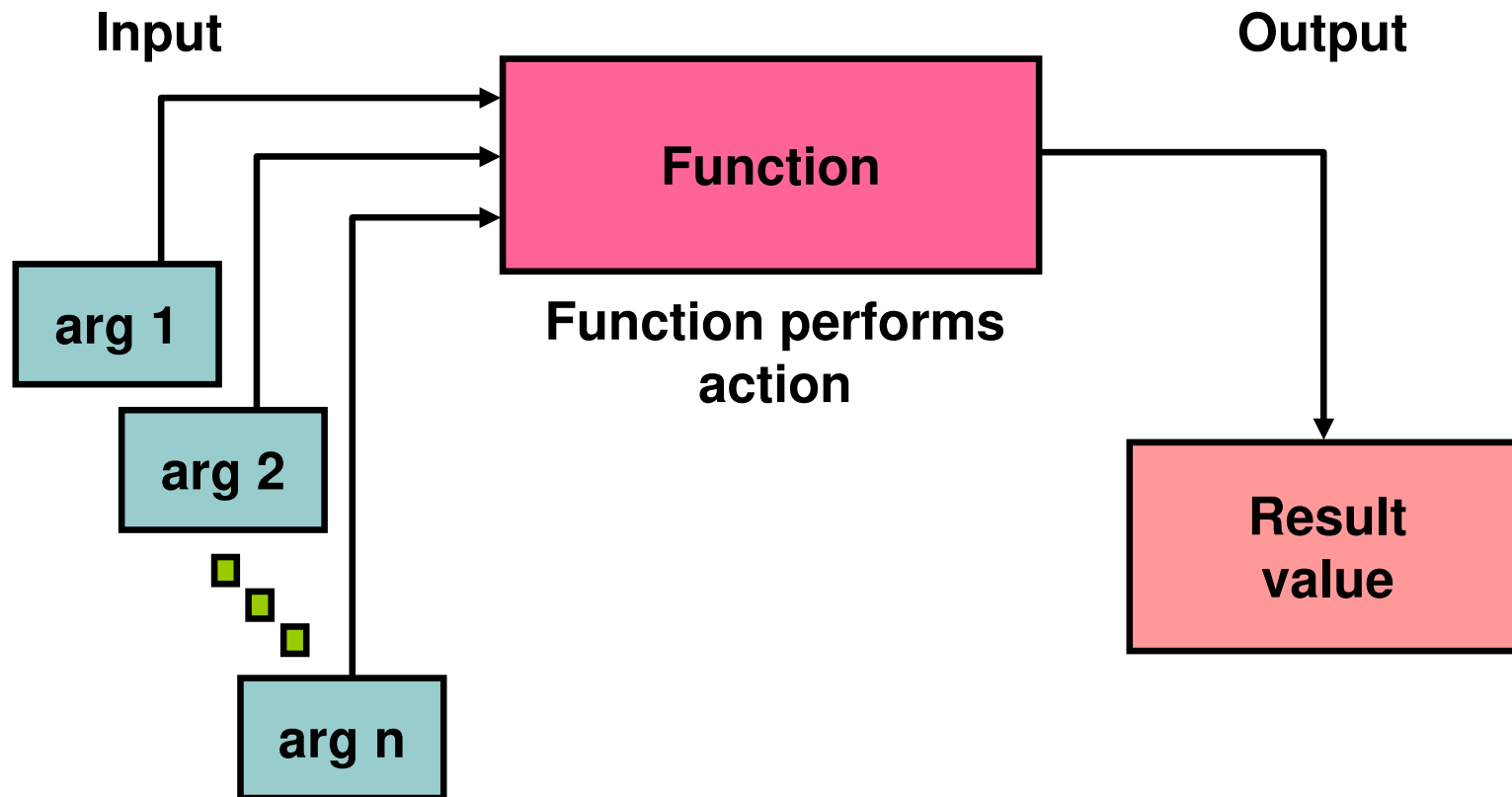
After completing this lesson, you should be able to do the following:

- Describe various types of functions available in SQL
- Use character, number, and date functions in `SELECT` statements

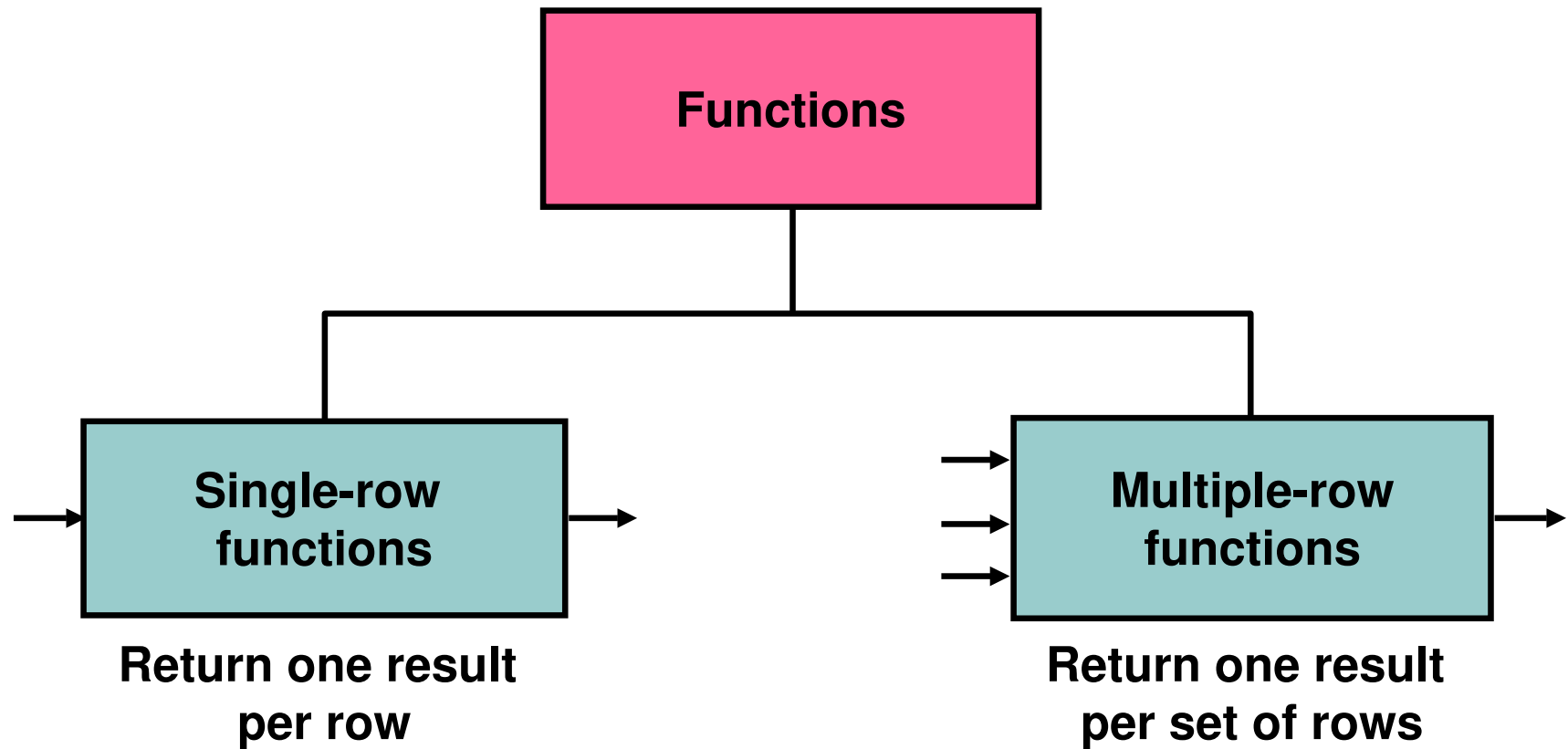
# Lesson Agenda

- Single-row SQL functions
- Character functions
- Number functions
- Working with dates
- Date functions

# SQL Functions



# Two Types of SQL Functions





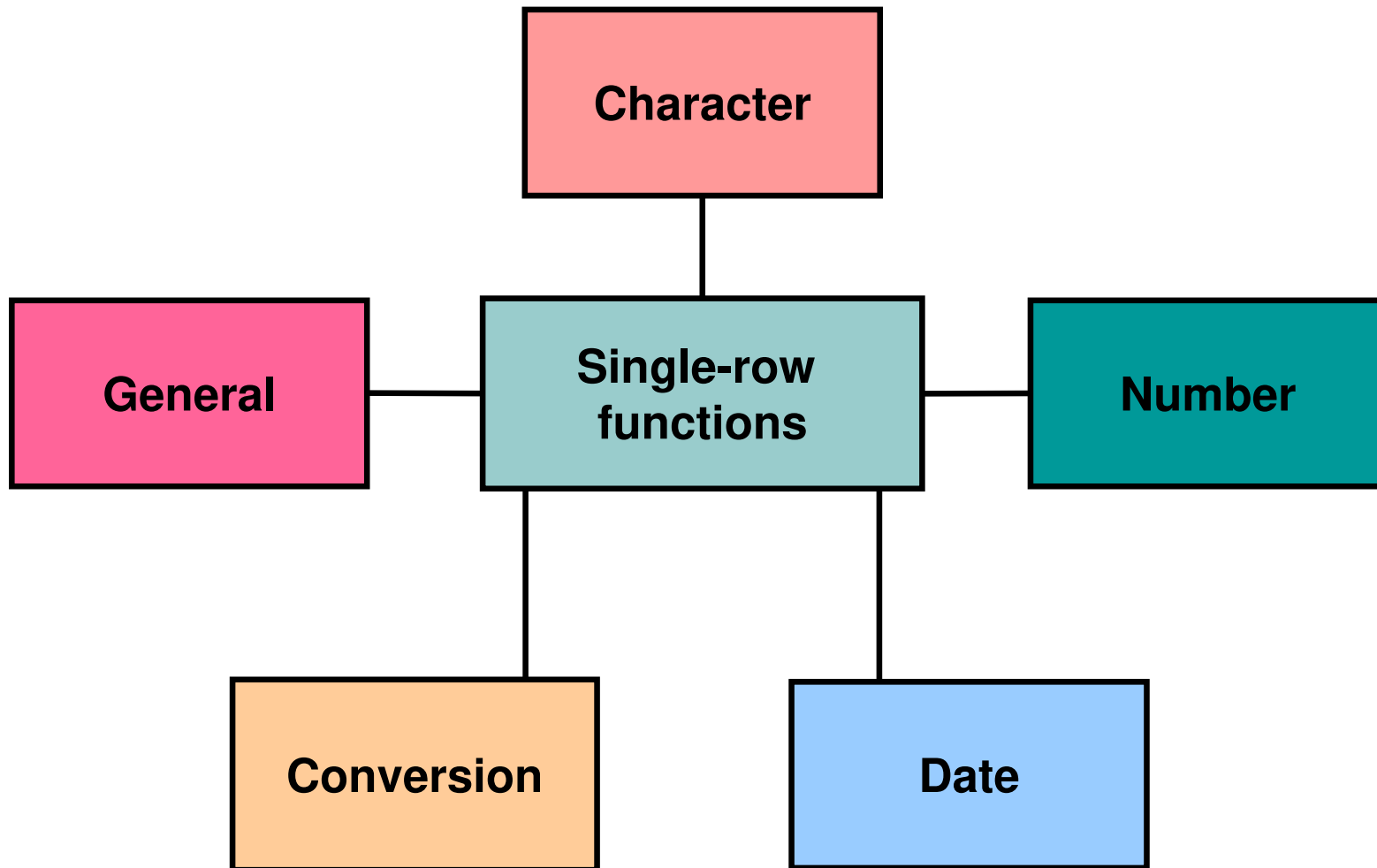
# Single-Row Functions

Single-row functions:

- Manipulate data items
- Accept arguments and return one value
- Act on each row that is returned
- Return one result per row
- May modify the data type
- Can be nested
- Accept arguments that can be a column or an expression

```
function_name [(arg1, arg2, ...)]
```

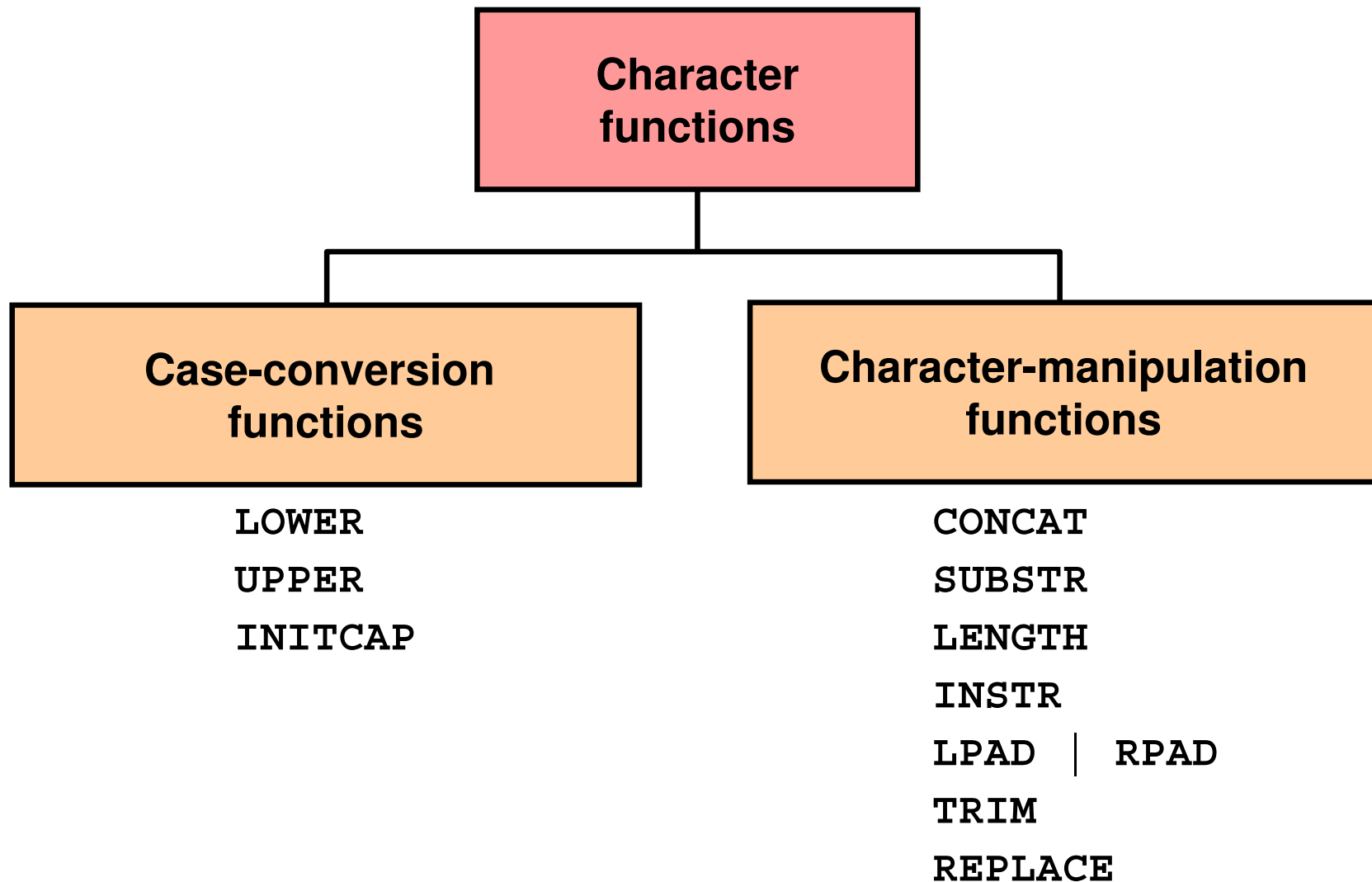
# Single-Row Functions



# Lesson Agenda

- Single-row SQL functions
- **Character functions**
- Number functions
- Working with dates
- Date functions

# Character Functions



# Case-Conversion Functions

These functions convert the case for character strings:

Function	Result
<code>LOWER('SQL Course')</code>	sql course
<code>UPPER('SQL Course')</code>	SQL COURSE
<code>INITCAP('SQL Course')</code>	Sql Course

# Using Case-Conversion Functions

Display the employee number, name, and department number for employee Higgins:

```
SELECT employee_id, last_name, department_id
FROM   employees
WHERE  last_name = 'higgins';
```

0 rows selected

```
SELECT employee_id, last_name, department_id
FROM   employees
WHERE  LOWER(last_name) = 'higgins';
```

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	205	Higgins	110

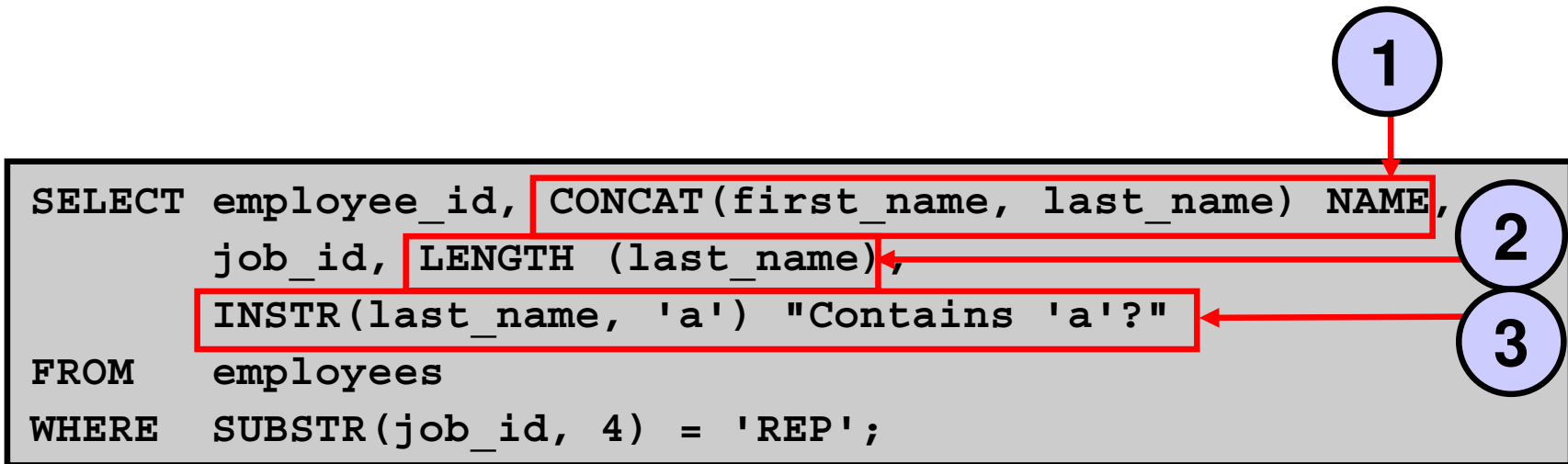
# Character-Manipulation Functions

These functions manipulate character strings:

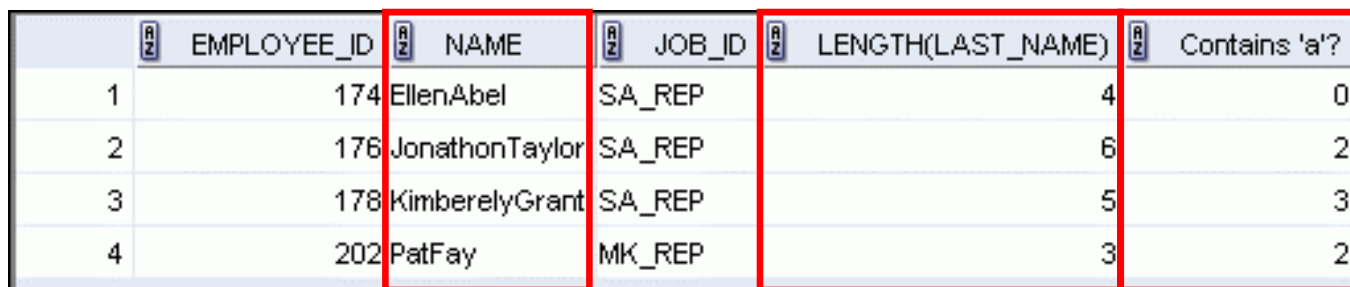
Function	Result
CONCAT('Hello', 'World')	HelloWorld
SUBSTR('HelloWorld',1,5)	Hello
LENGTH('HelloWorld')	10
INSTR('HelloWorld', 'W')	6
LPAD(salary,10,'*')	*****24000
RPAD(salary, 10, '*')	24000*****
REPLACE ('JACK and JUE','J','BL')	BLACK and BLUE
TRIM('H' FROM 'HelloWorld')	elloWorld

# Using the Character-Manipulation Functions

```
SELECT employee_id, CONCAT(first_name, last_name) NAME,  
       job_id, LENGTH (last_name),  
       INSTR(last_name, 'a') "Contains 'a'?"  
FROM   employees  
WHERE  SUBSTR(job_id, 4) = 'REP';
```



	EMPLOYEE_ID	NAME	JOB_ID	LENGTH(LAST_NAME)	Contains 'a'?
1	174	EllenAbel	SA_REP	4	0
2	176	JonathonTaylor	SA_REP	6	2
3	178	KimberelyGrant	SA_REP	5	3
4	202	PatFay	MK_REP	3	2





# Lesson Agenda

- Single-row SQL functions
- Character functions
- **Number functions**
- Working with dates
- Date Functions

# Number Functions

- ROUND: Rounds value to a specified decimal
- TRUNC: Truncates value to a specified decimal
- MOD: Returns remainder of division

Function	Result
ROUND (45.926, 2)	45.93
TRUNC (45.926, 2)	45.92
MOD (1600, 300)	100

# Using the ROUND Function

The diagram illustrates the use of the ROUND function in SQL. It shows a query and its output with numbered annotations:

- Annotation 1:** Points to the first ROUND function in the query: `ROUND(45.923, 2)`.
- Annotation 2:** Points to the second ROUND function in the query: `ROUND(45.923, 0)`.
- Annotation 3:** Points to the third ROUND function in the query: `ROUND(45.923, -1)`.

The query is:

```
SELECT ROUND(45.923, 2), ROUND(45.923, 0),  
       ROUND(45.923, -1)  
FROM   DUAL;
```

The results are shown in a table with one row and three columns, corresponding to the three ROUND functions. The first column shows the result of `ROUND(45.923, 2)` as 45.92. The second column shows the result of `ROUND(45.923, 0)` as 46. The third column shows the result of `ROUND(45.923, -1)` as 50.

	ROUND(45.923,2)	ROUND(45.923,0)	ROUND(45.923,-1)
1	45.92	46	50

DUAL is a dummy table that you can use to view results from functions and calculations.

# Using the TRUNC Function

1 2 3

```
SELECT TRUNC(45.923, 2), TRUNC(45.923),  
       TRUNC(45.923, -1)  
FROM DUAL;
```

	TRUNC(45.923,2)	TRUNC(45.923)	TRUNC(45.923,-1)
1	45.92	45	40

1 2 3

# Using the MOD Function

For all employees with the job title of Sales Representative, calculate the remainder of the salary after it is divided by 5,000.

```
SELECT last_name, salary, MOD(salary, 5000)
FROM   employees
WHERE  job_id = 'SA_REP';
```

	LAST_NAME	SALARY	MOD(SALARY,5000)
1	Abel	11000	1000
2	Taylor	8600	3600
3	Grant	7000	2000

# Lesson Agenda

- Single-row SQL functions
- Character functions
- Number functions
- **Working with dates**
- Date functions

# Working with Dates

- The Oracle database stores dates in an internal numeric format: century, year, month, day, hours, minutes, and seconds.
- The default date display format is DD-MON-RR.
  - Enables you to store 21st-century dates in the 20th century by specifying only the last two digits of the year
  - Enables you to store 20th-century dates in the 21st century in the same way

```
SELECT last_name, hire_date
FROM   employees
WHERE  hire_date < '01-FEB-88';
```

	LAST_NAME	HIRE_DATE
1	King	17-JUN-87
2	Whalen	17-SEP-87

# RR Date Format

Current Year	Specified Date	RR Format	YY Format
1995	27-OCT-95	1995	1995
1995	27-OCT-17	2017	1917
2001	27-OCT-17	2017	2017
2001	27-OCT-95	1995	2095

		If the specified two-digit year is:	
		0–49	50–99
If two digits of the current year are:	0–49	The return date is in the current century	The return date is in the century before the current one
	50–99	The return date is in the century after the current one	The return date is in the current century



# Using the SYSDATE Function

SYSDATE is a function that returns:

- Date
- Time

```
SELECT sysdate  
FROM dual;
```

	SYSDATE
1	31-MAY-07

# Arithmetic with Dates

- Add or subtract a number to or from a date for a resultant date value.
- Subtract two dates to find the number of days between those dates.
- Add hours to a date by dividing the number of hours by 24.

# Using Arithmetic Operators with Dates

```
SELECT last_name, (SYSDATE-hire_date)/7 AS WEEKS  
FROM employees  
WHERE department_id = 90;
```

	LAST_NAME	WEEKS
1	King	1041.168239087301587301587301587302
2	Kochhar	923.025381944444444444444444444444
3	De Haan	750.168239087301587301587301587302

# Lesson Agenda

- Single-row SQL functions
- Character functions
- Number functions
- Working with dates
- **Date functions**

# Date-Manipulation Functions

Function	Result
MONTHS_BETWEEN	Number of months between two dates
ADD_MONTHS	Add calendar months to date
NEXT_DAY	Next day of the date specified
LAST_DAY	Last day of the month
ROUND	Round date
TRUNC	Truncate date

# Using Date Functions

Function	Result
MONTHS_BETWEEN ( '01-SEP-95' , '11-JAN-94' )	19.6774194
ADD_MONTHS ( '31-JAN-96' , 1 )	'29-FEB-96'
NEXT_DAY ( '01-SEP-95' , 'FRIDAY' )	'08-SEP-95'
LAST_DAY ( '01-FEB-95' )	'28-FEB-95'

# Using ROUND and TRUNC Functions with Dates

Assume SYSDATE = '25-JUL-03':

Function	Result
ROUND (SYSDATE , 'MONTH' )	01-AUG-03
ROUND (SYSDATE , 'YEAR' )	01-JAN-04
TRUNC (SYSDATE , 'MONTH' )	01-JUL-03
TRUNC (SYSDATE , 'YEAR' )	01-JAN-03

# Quiz

Which of the following statements are true about single-row functions?

1. Manipulate data items
2. Accept arguments and return one value per argument
3. Act on each row that is returned
4. Return one result per set of rows
5. May not modify the data type
6. Can be nested
7. Accept arguments that can be a column or an expression



# Summary

In this lesson, you should have learned how to:

- Perform calculations on data using functions
- Modify individual data items using functions

## Practice 3: Overview

This practice covers the following topics:

- Writing a query that displays the current date
- Creating queries that require the use of numeric, character, and date functions
- Performing calculations of years and months of service for an employee



# **Using Conversion Functions and Conditional Expressions**

# Objectives

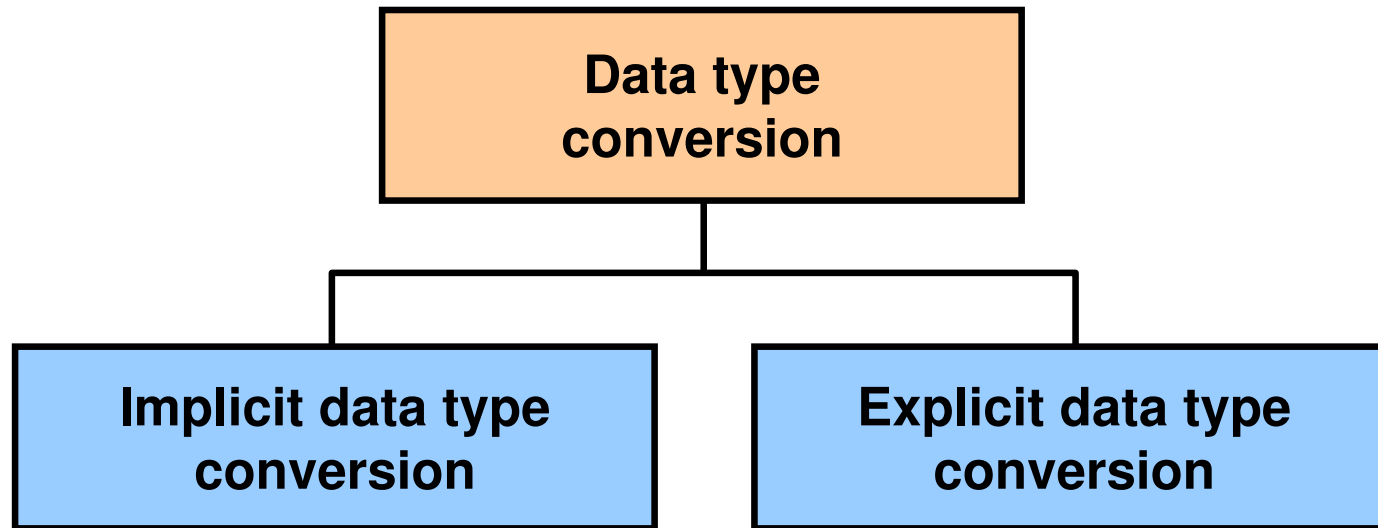
After completing this lesson, you should be able to do the following:

- Describe various types of conversion functions that are available in SQL
- Use the `TO_CHAR`, `TO_NUMBER`, and `TO_DATE` conversion functions
- Apply conditional expressions in a `SELECT` statement

# Lesson Agenda

- Implicit and explicit data type conversion
- TO\_CHAR, TO\_DATE, TO\_NUMBER functions
- Nesting functions
- General functions:
  - NVL
  - NVL2
  - NULLIF
  - COALESCE
- Conditional expressions:
  - CASE
  - DECODE

# Conversion Functions



# Implicit Data Type Conversion

In expressions, the Oracle server can automatically convert the following:

From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE

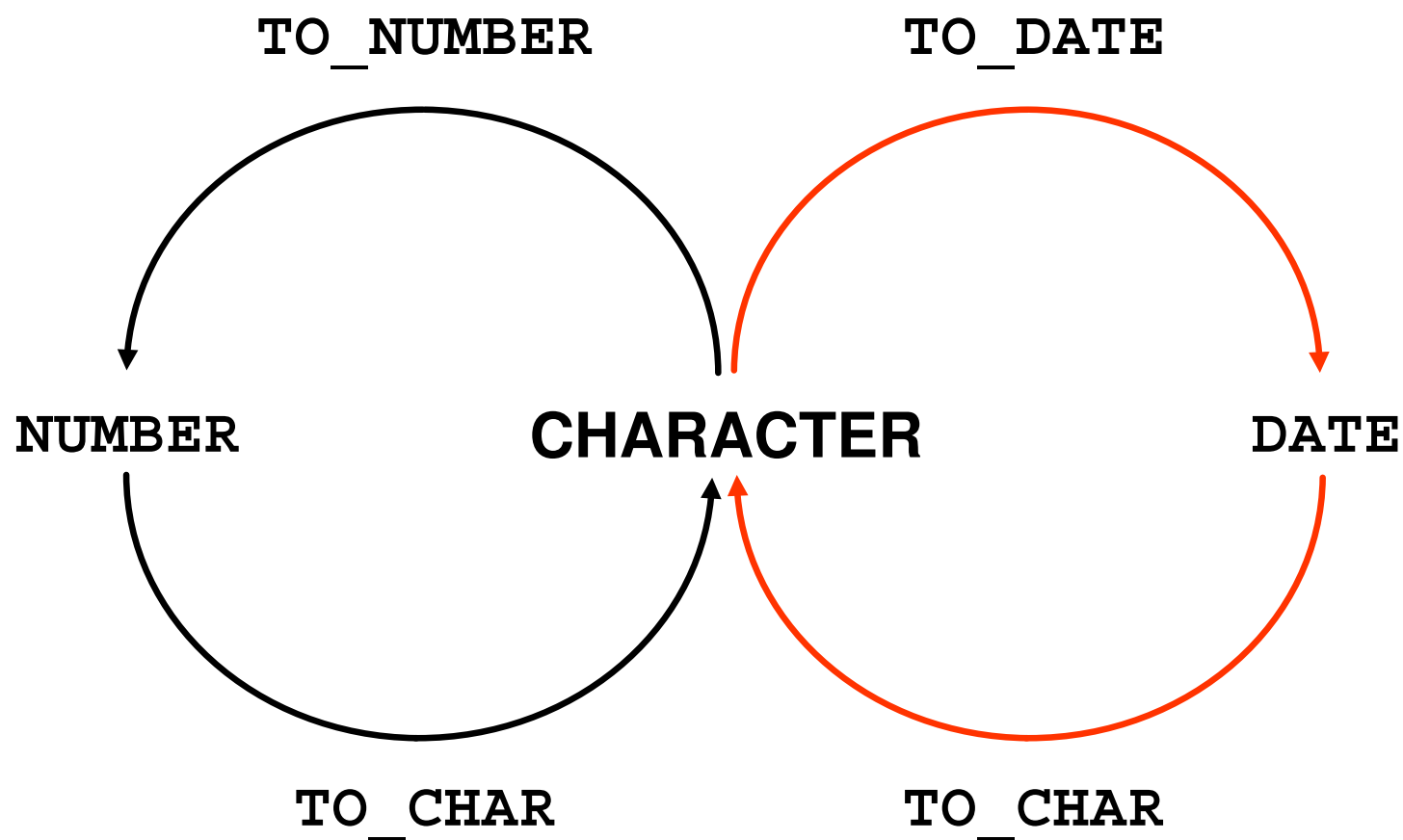
# Implicit Data Type Conversion

For expression evaluation, the Oracle server can automatically convert the following:

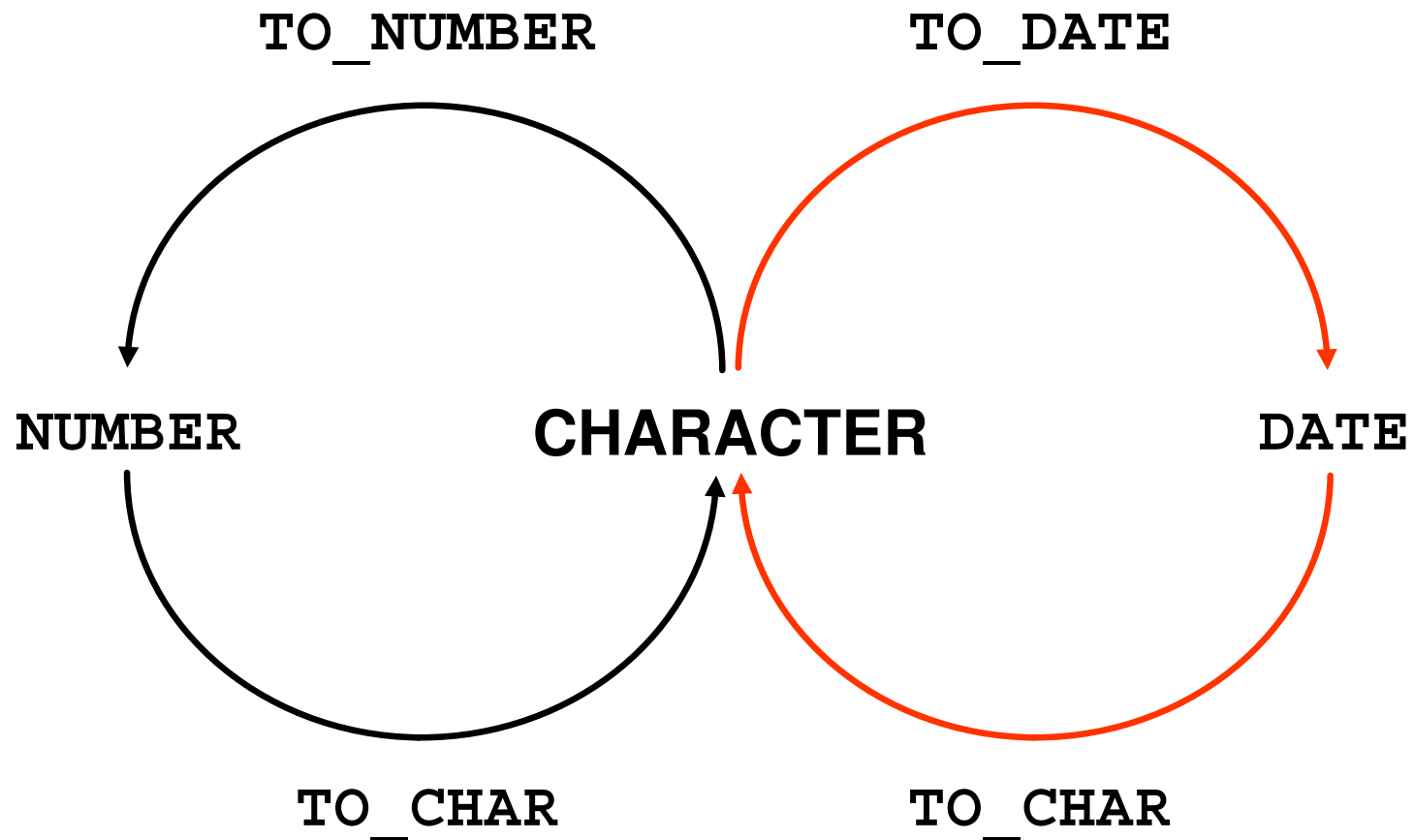
From	To
NUMBER	VARCHAR2 or CHAR
DATE	VARCHAR2 or CHAR



# Explicit Data Type Conversion



# Explicit Data Type Conversion



# Lesson Agenda

- Implicit and explicit data type conversion
- **TO\_CHAR, TO\_DATE, TO\_NUMBER functions**
- Nesting functions
- General functions:
  - NVL
  - NVL2
  - NULLIF
  - COALESCE
- Conditional expressions:
  - CASE
  - DECODE

# Using the TO\_CHAR Function with Dates

```
TO_CHAR(date, 'format_model')  

```

The format model:

- Must be enclosed with single quotation marks
- Is case-sensitive
- Can include any valid date format element
- Has an `fm` element to remove padded blanks or suppress leading zeros
- Is separated from the date value by a comma

# Elements of the Date Format Model

Element	Result
YYYY	Full year in numbers
YEAR	Year spelled out (in English)
MM	Two-digit value for the month
MONTH	Full name of the month
MON	Three-letter abbreviation of the month
DY	Three-letter abbreviation of the day of the week
DAY	Full name of the day of the week
DD	Numeric day of the month

# Elements of the Date Format Model

- Time elements format the time portion of the date:

HH24:MI:SS AM	15:45:32 PM
---------------	-------------

- Add character strings by enclosing them with double quotation marks:

DD "of" MONTH	12 of OCTOBER
---------------	---------------

- Number suffixes spell out numbers:

ddspth	fourteenth
--------	------------

# Using the TO\_CHAR Function with Dates

```
SELECT last_name,  
       TO_CHAR(hire_date, 'fmDD Month YYYY')  
       AS HIREDATE  
FROM   employees;
```

	LAST_NAME	HIREDATE
1	King	17 June 1987
2	Kochhar	21 September 1989
3	De Haan	13 January 1993
4	Hunold	3 January 1990
5	Ernst	21 May 1991
6	Lorentz	7 February 1999
7	Mourgos	16 November 1999
8	Rajs	17 October 1995
9	Davies	29 January 1997
10	Matos	15 March 1998
...		
19	Higgins	7 June 1994
20	Gietz	7 June 1994

# Using the TO\_CHAR Function with Numbers

```
TO_CHAR(number, 'format_model') 
```

These are some of the format elements that you can use with the TO\_CHAR function to display a number value as a character:

Element	Result
9	Represents a number
0	Forces a zero to be displayed
\$	Places a floating dollar sign
L	Uses the floating local currency symbol
.	Prints a decimal point
,	Prints a comma as a thousands indicator



# Using the TO\_CHAR Function with Numbers

```
SELECT TO_CHAR(salary, '$99,999.00') SALARY  
FROM   employees  
WHERE  last_name = 'Ernst';
```

	SALARY
1	\$6,000.00

# Using the TO\_NUMBER and TO\_DATE Functions

- Convert a character string to a number format using the TO\_NUMBER function:

```
TO_NUMBER(char[, 'format_model'])
```

- Convert a character string to a date format using the TO\_DATE function:

```
TO_DATE(char[, 'format_model'])
```

- These functions have an `fx` modifier. This modifier specifies the exact match for the character argument and date format model of a TO\_DATE function.

# Using the TO\_CHAR and TO\_DATE Function with RR Date Format

To find employees hired before 1990, use the RR date format, which produces the same results whether the command is run in 1999 or now:

```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-YYYY')
FROM employees
WHERE hire_date < TO_DATE('01-Jan-90', 'DD-Mon-RR');
```

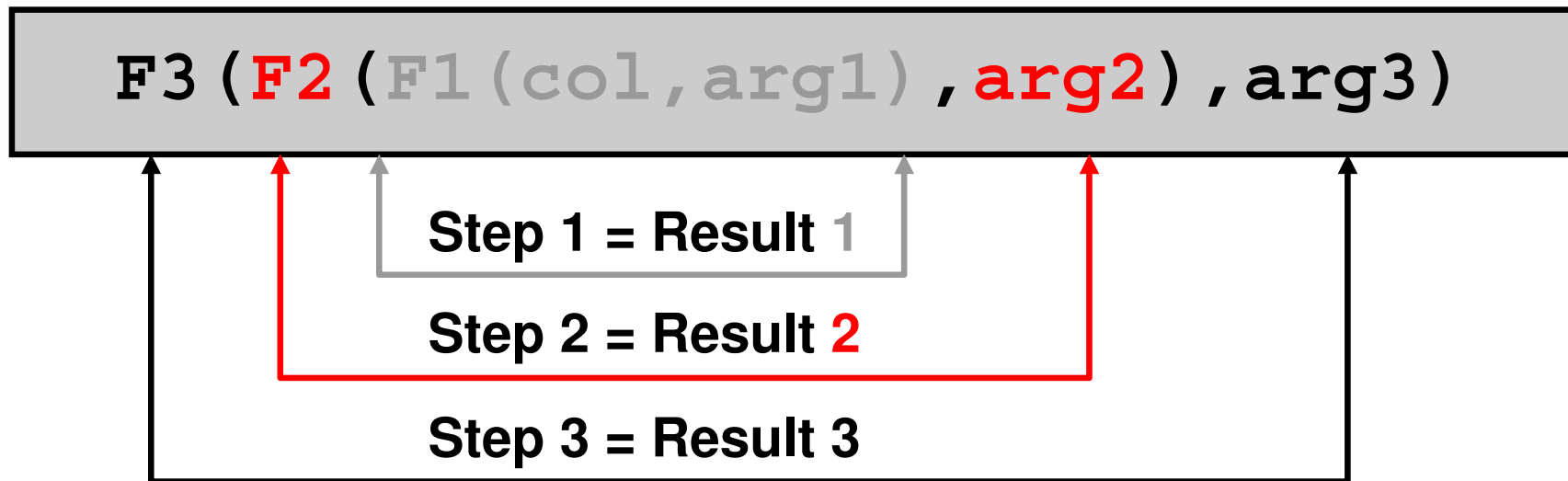
	LAST_NAME	TO_CHAR(HIRE_DATE,'DD-MON-YYYY')
1	King	17-Jun-1987
2	Kochhar	21-Sep-1989
3	Whalen	17-Sep-1987

# Lesson Agenda

- Implicit and explicit data type conversion
- TO\_CHAR, TO\_DATE, TO\_NUMBER functions
- **Nesting functions**
- General functions:
  - NVL
  - NVL2
  - NULLIF
  - COALESCE
- Conditional expressions:
  - CASE
  - DECODE

# Nesting Functions

- Single-row functions can be nested to any level.
- Nested functions are evaluated from the deepest level to the least deep level.



# Nesting Functions

```
SELECT last_name,  
       UPPER(CONCAT(SUBSTR (LAST_NAME, 1, 8), '_US'))  
FROM   employees  
WHERE  department_id = 60;
```

	LAST_NAME	UPPER(CONCAT(SUBSTR(LAST_NAME,1,8),'_US'))
1	Hunold	HUNOLD_US
2	Ernst	ERNST_US
3	Lorentz	LORENTZ_US

# Lesson Agenda

- Implicit and explicit data type conversion
- TO\_CHAR, TO\_DATE, TO\_NUMBER functions
- Nesting functions
- **General functions:**
  - NVL
  - NVL2
  - NULLIF
  - COALESCE
- Conditional expressions:
  - CASE
  - DECODE

# General Functions

The following functions work with any data type and pertain to using nulls:

- `NVL (expr1, expr2)`
- `NVL2 (expr1, expr2, expr3)`
- `NULLIF (expr1, expr2)`
- `COALESCE (expr1, expr2, ..., exprn)`



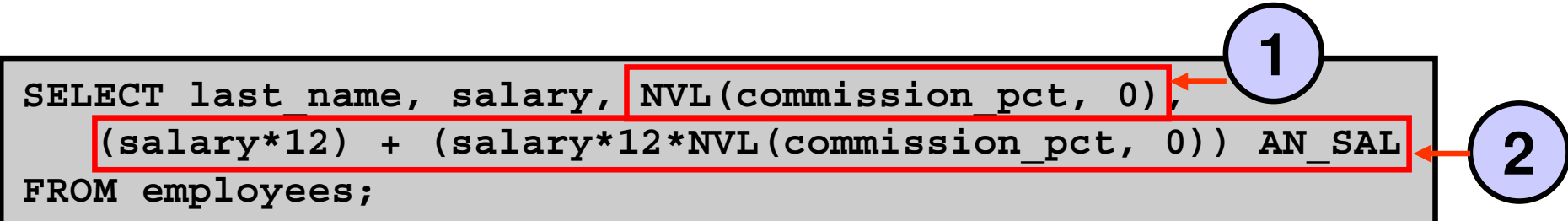
# NVL Function

Converts a null value to an actual value:

- Data types that can be used are date, character, and number.
- Data types must match:
  - `NVL(commission_pct, 0)`
  - `NVL(hire_date, '01-JAN-97')`
  - `NVL(job_id, 'No Job Yet')`

# Using the NVL Function

```
SELECT last_name, salary, NVL(commission_pct, 0),  
       (salary*12) + (salary*12*NVL(commission_pct, 0)) AN_SAL  
FROM employees;
```



	LAST_NAME	SALARY	NVL(COMMISSION_PCT,0)	AN_SAL
1	King	24000	0	288000
2	Kochhar	17000	0	204000
3	De Haan	17000	0	204000
4	Hunold	9000	0	108000
5	Ernst	6000	0	72000
6	Lorentz	4200	0	50400
7	Mourgos	5800	0	69600
8	Rajs	3500	0	42000
9	Davies	3100	0	37200
10	Matos	2600	0	31200
11	Vargas	2500	0	30000
12	Zlotkey	10500	0.2	151200

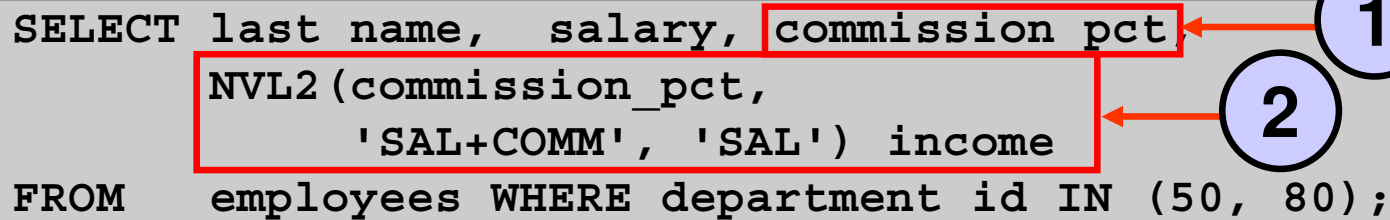
...

1

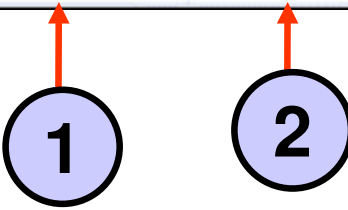
2

# Using the NVL2 Function

```
SELECT last name, salary, commission_pct  
      NVL2(commission_pct,  
            'SAL+COMM', 'SAL') income  
FROM   employees WHERE department_id IN (50, 80);
```

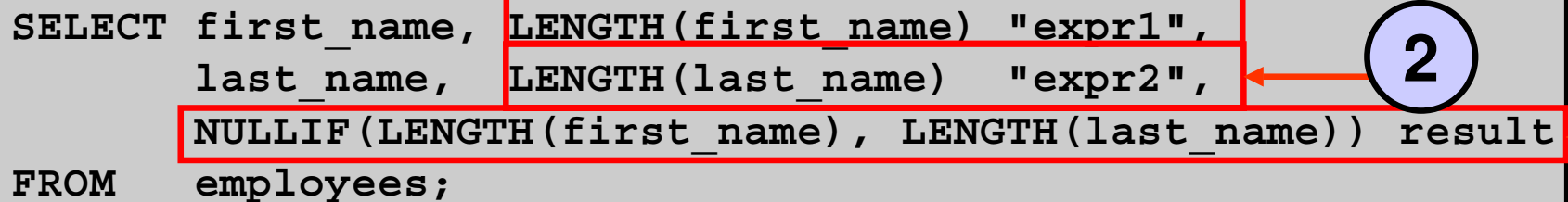


	LAST_NAME	SALARY	COMMISSION_PCT	INCOME
1	Mourgos	5800	(null)	SAL
2	Rajs	3500	(null)	SAL
3	Davies	3100	(null)	SAL
4	Matos	2600	(null)	SAL
5	Vargas	2500	(null)	SAL
6	Zlotkey	10500	0.2	SAL+COMM
7	Abel	11000	0.3	SAL+COMM
8	Taylor	8600	0.2	SAL+COMM



# Using the NULLIF Function


```
SELECT first_name, LENGTH(first_name) "expr1",  
       last_name,  LENGTH(last_name)  "expr2",  
       NULLIF(LENGTH(first_name), LENGTH(last_name)) result  
FROM employees;
```



	FIRST_NAME	expr1	LAST_NAME	expr2	RESULT
1	Ellen	5	Abel	4	5
2	Curtis	6	Davies	6	(null)
3	Lex	3	De Haan	7	3
4	Bruce	5	Ernst	5	(null)
5	Pat	3	Fay	3	(null)
6	William	7	Gietz	5	7
7	Kimberely	9	Grant	5	9

...

19	Jennifer	8	Whalen	6	8
20	Eleni	5	Zlotkey	7	5



# Using the COALESCE Function

- The advantage of the COALESCE function over the NVL function is that the COALESCE function can take multiple alternate values.
- If the first expression is not null, the COALESCE function returns that expression; otherwise, it does a COALESCE of the remaining expressions.

# Using the COALESCE Function

```
SELECT last name, employee id,  
COALESCE(TO_CHAR(commission_pct), TO_CHAR(manager_id),  
         'No commission and no manager')  
FROM employees;
```

	LAST_NAME	EMPLOYEE_ID	COALESCE(TO_CHAR(COMMISSION_PCT), TO_CHAR(MANAGER_ID), 'No commission and no manager')
1	King	100	No commission and no manager
2	Kochhar	101	100
3	De Haan	102	100
4	Hunold	103	102
5	Ernst	104	103
6	Lorentz	107	103
7	Mourgos	124	100
8	Rajs	141	124

...

12	Zlotkey	149	.2
13	Abel	174	.3
14	Taylor	176	.2
15	Grant	178	.15
16	Whalen	200	101

...

# Lesson Agenda

- Implicit and explicit data type conversion
- TO\_CHAR, TO\_DATE, TO\_NUMBER functions
- Nesting functions
- General functions:
  - NVL
  - NVL2
  - NULLIF
  - COALESCE
- Conditional expressions:
  - CASE
  - DECODE

# Conditional Expressions

- Provide the use of the `IF-THEN-ELSE` logic within a SQL statement
- Use two methods:
  - `CASE` expression
  - `DECODE` function



# CASE Expression

Facilitates conditional inquiries by doing the work of an IF-THEN-ELSE statement:

```
CASE expr WHEN comparison_expr1 THEN return_expr1  
      [WHEN comparison_expr2 THEN return_expr2  
      WHEN comparison_exprn THEN return_exprn  
      ELSE else_expr]  
END
```

# Using the CASE Expression

Facilitates conditional inquiries by doing the work of an IF-THEN-ELSE statement:

```
SELECT last_name, job_id, salary,  
       CASE job_id WHEN 'IT_PROG' THEN 1.10*salary  
                  WHEN 'ST_CLERK' THEN 1.15*salary  
                  WHEN 'SA_REP' THEN 1.20*salary  
       ELSE salary END "REVISED_SALARY"  
FROM employees;
```

	LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
...				
5	Ernst	IT_PROG	6000	6600
6	Lorentz	IT_PROG	4200	4620
7	Mourgos	ST_MAN	5800	5800
8	Rajs	ST_CLERK	3500	4025
9	Davies	ST_CLERK	3100	3565
...				
13	Abel	SA_REP	11000	13200
14	Taylor	SA_REP	8600	10320
...				

# DECODE Function

Facilitates conditional inquiries by doing the work of a CASE expression or an IF-THEN-ELSE statement:

```
DECODE(col/expression, search1, result1  
      [, search2, result2, ...,]  
      [, default])
```

# Using the DECODE Function

```
SELECT last name, job id, salary,  
       DECODE(job_id, 'IT_PROG', 1.10*salary,  
                 'ST_CLERK', 1.15*salary,  
                 'SA_REP', 1.20*salary,  
                 salary)  
       REVISED_SALARY  
FROM   employees;
```

	LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
...				
6	Lorentz	IT_PROG	4200	4620
7	Mourgos	ST_MAN	5800	5800
8	Rajs	ST_CLERK	3500	4025
...				
13	Abel	SA_REP	11000	13200
14	Taylor	SA_REP	8600	10320
...				

# Using the DECODE Function

Display the applicable tax rate for each employee in department 80:

```
SELECT last_name, salary,  
       DECODE (TRUNC(salary/2000, 0),  
               0, 0.00,  
               1, 0.09,  
               2, 0.20,  
               3, 0.30,  
               4, 0.40,  
               5, 0.42,  
               6, 0.44,  
               0.45) TAX_RATE  
FROM   employees  
WHERE  department_id = 80;
```

# Quiz

The `TO_NUMBER` function converts either character strings or date values to a number in the format specified by the optional format model.

1. True
2. False

# Summary

In this lesson, you should have learned how to:

- Alter date formats for display using functions
- Convert column data types using functions
- Use NVL functions
- Use IF-THEN-ELSE logic and other conditional expressions in a SELECT statement

# Practice 4: Overview

This practice covers the following topics:

- Creating queries that use TO\_CHAR, TO\_DATE, and other DATE functions
- Creating queries that use conditional expressions such as DECODE and CASE





# **Reporting Aggregated Data Using the Group Functions**

# Objectives

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

- Identify the available group functions
- Describe the use of group functions
- Group data by using the `GROUP BY` clause
- Include or exclude grouped rows by using the `HAVING` clause

# Lesson Agenda

- Group functions:
  - Types and syntax
  - Use AVG, SUM, MIN, MAX, COUNT
  - Use DISTINCT keyword within group functions
  - NULL values in a group function
- Grouping rows:
  - GROUP BY clause
  - HAVING clause
- Nesting group functions

# What Are Group Functions?

Group functions operate on sets of rows to give one result per group.

## EMPLOYEES

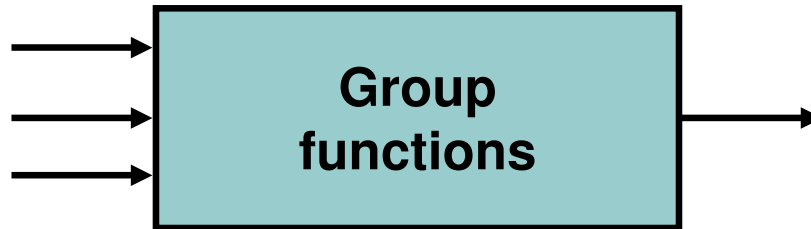
	DEPARTMENT_ID	SALARY
1	90	24000
2	90	17000
3	90	17000
4	60	9000
5	60	6000
6	60	4200
7	50	5800
8	50	3500
9	50	3100
10	50	2600
...		
18	20	6000
19	110	12000
20	110	8300

**Maximum salary in  
EMPLOYEES table**

MAX(SALARY)
24000

# Types of Group Functions

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



# Group Functions: Syntax

```
SELECT    group_function(column), ...
FROM      table
[WHERE    condition]
[ORDER BY column];
```

# Using the AVG and SUM Functions

You can use AVG and SUM for numeric data.

```
SELECT AVG(salary), MAX(salary),  
       MIN(salary), SUM(salary)  
FROM   employees  
WHERE  job_id LIKE '%REP%';
```

	AVG(SALARY)	MAX(SALARY)	MIN(SALARY)	SUM(SALARY)
1	8150	11000	6000	32600

# Using the MIN and MAX Functions

You can use **MIN** and **MAX** for numeric, character, and date data types.

```
SELECT MIN(hire_date), MAX(hire_date)
FROM   employees;
```

	MIN(HIRE_DATE)	MAX(HIRE_DATE)
1	17-JUN-87	29-JAN-00



# Using the COUNT Function

COUNT ( \* ) returns the number of rows in a table:

1

```
SELECT COUNT ( * )  
FROM   employees  
WHERE  department_id = 50;
```

	COUNT(*)
1	5

COUNT ( *expr* ) returns the number of rows with non-null values for *expr*:

2

```
SELECT COUNT (commission_pct)  
FROM   employees  
WHERE  department_id = 80;
```

	COUNT(COMMISSION_PCT)
1	3

# Using the DISTINCT Keyword

- COUNT (DISTINCT expr) returns the number of distinct non-null values of *expr*.
- To display the number of distinct department values in the EMPLOYEES table:

```
SELECT COUNT(DISTINCT department_id)  
FROM employees;
```

	1	2	COUNT(DISTINCTDEPARTMENT_ID)
	1		7

# Group Functions and Null Values

Group functions ignore null values in the column:

1

```
SELECT AVG(commission_pct)
FROM employees;
```

	Avg(COMMISSION_PCT)
1	0.2125

The NVL function forces group functions to include null values:

2

```
SELECT AVG(NVL(commission_pct, 0))
FROM employees;
```

	Avg(NVL(COMMISSION_PCT,0))
1	0.0425

# Lesson Agenda

- Group functions:
  - Types and syntax
  - Use AVG, SUM, MIN, MAX, COUNT
  - Use DISTINCT keyword within group functions
  - NULL values in a group function
- Grouping rows:
  - GROUP BY clause
  - HAVING clause
- Nesting group functions

# Creating Groups of Data

## EMPLOYEES

	DEPARTMENT_ID	SALARY
1	10	4400
2	20	13000
3	20	6000
4	50	5800
5	50	2500
6	50	2600
7	50	3100
8	50	3500
9	60	4200
10	60	6000
11	60	9000
12	80	11000
13	80	10500
14	80	8600
...		
19	110	12000
20	(null)	7000

4400

9500

3500

6400

10033

**Average salary in  
EMPLOYEES table for  
each department**

	DEPARTMENT_ID	AVG(SALARY)
1	10	4400
2	20	9500
3	50	3500
4	60	6400
5	80	10033.333333333333...
6	90	19333.333333333333...
7	110	10150
8	(null)	7000

# Creating Groups of Data: GROUP BY Clause Syntax

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

You can divide rows in a table into smaller groups by using the GROUP BY clause.

# Using the GROUP BY Clause

All columns in the SELECT list that are not in group functions must be in the GROUP BY clause.

```
SELECT department_id, AVG(salary)
FROM employees
GROUP BY department_id ;
```

	DEPARTMENT_ID	AVG(SALARY)
1	(null)	7000
2	90	19333.3333333333...
3	20	9500
4	110	10150
5	50	3500
6	80	10033.3333333333...
7	60	6400
8	10	4400

# Using the GROUP BY Clause

The GROUP BY column does not have to be in the SELECT list.

```
SELECT    AVG(salary)
FROM      employees
GROUP BY  department_id ;
```

	AVG(SALARY)
1	7000
2	19333.333333333333333333333333...
3	9500
4	10150
5	3500
6	10033.333333333333333333333333...
7	6400
8	4400



# Grouping by More than One Column

## EMPLOYEES

	DEPARTMENT_ID	JOB_ID	SALARY
1	10	AD_ASST	4400
2	20	MK_MAN	13000
3	20	MK_REP	6000
4	50	ST_MAN	5800
5	50	ST_CLERK	2500
6	50	ST_CLERK	2600
7	50	ST_CLERK	3100
8	50	ST_CLERK	3500
9	60	IT_PROG	4200
10	60	IT_PROG	6000
11	60	IT_PROG	9000
12	80	SA_REP	11000
13	80	SA_MAN	10500
14	80	SA_REP	8600
...			
19	110	AC_MGR	12000
20	(null)	SA_REP	7000

Add the salaries in the **EMPLOYEES** table for each job, grouped by department.

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	10	AD_ASST	4400
2	20	MK_MAN	13000
3	20	MK_REP	6000
4	50	ST_CLERK	11700
5	50	ST_MAN	5800
6	60	IT_PROG	19200
7	80	SA_MAN	10500
8	80	SA_REP	19600
9	90	AD_PRES	24000
10	90	AD_VP	34000
11	110	AC_ACCOUNT	8300
12	110	AC_MGR	12000
13	(null)	SA_REP	7000

# Using the GROUP BY Clause on Multiple Columns

```
SELECT department_id, job_id, SUM(salary)
FROM employees
WHERE department_id > 40
GROUP BY department_id, job_id
ORDER BY department_id;
```

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	50	ST_CLERK	11700
2	50	ST_MAN	5800
3	60	IT_PROG	19200
4	80	SA_MAN	10500
5	80	SA_REP	19600
6	90	AD_PRES	24000
7	90	AD_VP	34000
8	110	AC_ACCOUNT	8300
9	110	AC_MGR	12000

# Illegal Queries

## Using Group Functions

Any column or expression in the `SELECT` list that is not an aggregate function must be in the `GROUP BY` clause:

```
SELECT department_id, COUNT(last_name)
FROM   employees;
```

ORA-00937: not a single-group group function  
00937.00000 - "not a single-group group function"

**A `GROUP BY` clause must be added to count the last names for each `department_id`.**

```
SELECT department_id, job_id, COUNT(last_name)
FROM   employees
GROUP BY department_id;
```

ORA-00979: not a GROUP BY expression  
00979.00000 - "not a GROUP BY expression"

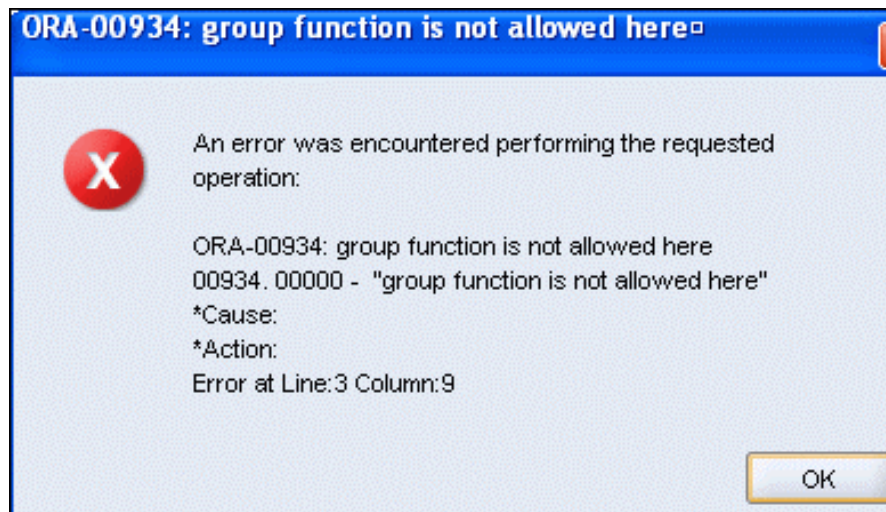
**Either add `job_id` in the `GROUP BY` or remove the `job_id` column from the `SELECT` list.**

# Illegal Queries

## Using Group Functions

- You cannot use the `WHERE` clause to restrict groups.
- You use the `HAVING` clause to restrict groups.
- You cannot use group functions in the `WHERE` clause.

```
SELECT    department_id, AVG(salary)
FROM      employees
WHERE     AVG(salary) > 8000
GROUP BY  department_id;
```



**Cannot use the  
`WHERE` clause to  
restrict groups**

# Restricting Group Results

## EMPLOYEES

	DEPARTMENT_ID	SALARY
1	10	4400
2	20	13000
3	20	6000
4	50	5800
5	50	2500
6	50	2600
7	50	3100
8	50	3500
9	60	4200
10	60	6000
11	60	9000
12	80	11000
13	80	10500
14	80	8600

...

18	110	8300
19	110	12000
20	(null)	7000

The maximum salary per department when it is greater than \$10,000

	DEPARTMENT_ID	MAX(SALARY)
1	20	13000
2	80	11000
3	90	24000
4	110	12000

# Restricting Group Results with the HAVING Clause

When you use the HAVING clause, the Oracle server restricts groups as follows:

1. Rows are grouped.
2. The group function is applied.
3. Groups matching the HAVING clause are displayed.

```
SELECT      column, group_function
FROM        table
[WHERE      condition]
[GROUP BY  group_by_expression]
[HAVING     group_condition]
[ORDER BY  column];
```



# Using the HAVING Clause

```
SELECT    department_id, MAX(salary)
FROM      employees
GROUP BY  department_id
HAVING    MAX(salary) > 10000 ;
```

	DEPARTMENT_ID	MAX(SALARY)
1	90	24000
2	20	13000
3	110	12000
4	80	11000

# Using the HAVING Clause

```
SELECT    job_id, SUM(salary) PAYROLL
FROM      employees
WHERE     job_id NOT LIKE '%REP%'
GROUP BY  job_id
HAVING    SUM(salary) > 13000
ORDER BY  SUM(salary);
```

	 JOB_ID	 PAYROLL
1	IT_PROG	19200
2	AD_PRES	24000
3	AD_VP	34000



# Lesson Agenda

- Group functions:
  - Types and syntax
  - Use AVG, SUM, MIN, MAX, COUNT
  - Use DISTINCT keyword within group functions
  - NULL values in a group function
- Grouping rows:
  - GROUP BY clause
  - HAVING clause
- Nesting group functions

# Nesting Group Functions

Display the maximum average salary:

```
SELECT MAX (AVG (salary))
FROM employees
GROUP BY department id;
```

[illegible]

# Quiz

Identify the guidelines for group functions and the GROUP BY clause.

1. You cannot use a column alias in the GROUP BY clause.
2. The GROUP BY column must be in the SELECT clause.
3. By using a WHERE clause, you can exclude rows before dividing them into groups.
4. The GROUP BY clause groups rows and ensures order of the result set.
5. If you include a group function in a SELECT clause, you cannot select individual results as well.

# Summary

In this lesson, you should have learned how to:

- Use the group functions COUNT, MAX, MIN, SUM, and AVG
- Write queries that use the GROUP BY clause
- Write queries that use the HAVING clause

```
SELECT      column, group_function
FROM        table
[WHERE      condition]
[GROUP BY  group_by_expression]
[HAVING     group_condition]
[ORDER BY  column];
```

# Practice 5: Overview

This practice covers the following topics:

- Writing queries that use the group functions
- Grouping by rows to achieve more than one result
- Restricting groups by using the `HAVING` clause



# **Displaying Data from Multiple Tables**

# Objectives

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

- Write `SELECT` statements to access data from more than one table using equijoins and nonequijoins
- Join a table to itself by using a self-join
- View data that generally does not meet a join condition by using `OUTER` joins
- Generate a Cartesian product of all rows from two or more tables

# Lesson Agenda

- Types of JOINS and its syntax
- Natural join:
  - USING clause
  - ON clause
- Self-join
- Nonequijoins
- OUTER join:
  - LEFT OUTER join
  - RIGHT OUTER join
  - FULL OUTER join
- Cartesian product
  - Cross join




# Obtaining Data from Multiple Tables

**EMPLOYEES**

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	100	King	90
2	101	Kochhar	90
3	102	De Haan	90
...			
18	202	Fay	20
19	205	Higgins	110
20	206	Gietz	110

**DEPARTMENTS**

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
1	10	Administration	1700
2	20	Marketing	1800
3	50	Shipping	1500
4	60	IT	1400
5	80	Sales	2500
6	90	Executive	1700
7	110	Accounting	1700
8	190	Contracting	1700



	EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
1	200	10	Administration
2	201	20	Marketing
3	202	20	Marketing
4	124	50	Shipping
5	144	50	Shipping
...			
18	205	110	Accounting
19	206	110	Accounting

# Types of Joins

Joins that are compliant with the SQL:1999 standard include the following:

- Natural joins:
  - NATURAL JOIN clause
  - USING clause
  - ON clause
- OUTER joins:
  - LEFT OUTER JOIN
  - RIGHT OUTER JOIN
  - FULL OUTER JOIN
- Cross joins

# Joining Tables Using SQL:1999 Syntax

Use a join to query data from more than one table:

```
SELECT    table1.column, table2.column
FROM      table1
[NATURAL JOIN table2] |
[JOIN table2 USING (column_name)] |
[JOIN table2
  ON (table1.column_name = table2.column_name)] |
[LEFT|RIGHT|FULL OUTER JOIN table2
  ON (table1.column_name = table2.column_name)] |
[CROSS JOIN table2];
```

# Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Use table prefixes to improve performance.
- Instead of full table name prefixes, use table aliases.
- Table alias gives a table a shorter name:
  - Keeps SQL code smaller, uses less memory
- Use column aliases to distinguish columns that have identical names, but reside in different tables.

# Lesson Agenda

- Types of JOINS and its syntax
- Natural join:
  - USING clause
  - ON clause
- Self-join
- Nonequijoins
- OUTER join:
  - LEFT OUTER join
  - RIGHT OUTER join
  - FULL OUTER join
- Cartesian product
  - Cross join

# Creating Natural Joins

- The NATURAL JOIN clause is based on all columns in the two tables that have the same name.
- It selects rows from the two tables that have equal values in all matched columns.
- If the columns having the same names have different data types, an error is returned.

# Retrieving Records with Natural Joins

```
SELECT department_id, department_name,  
       location_id, city  
FROM   departments  
NATURAL JOIN locations ;
```

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID	CITY
1	60	IT	1400	Southlake
2	50	Shipping	1500	South San Francisco
3	10	Administration	1700	Seattle
4	90	Executive	1700	Seattle
5	110	Accounting	1700	Seattle
6	190	Contracting	1700	Seattle
7	20	Marketing	1800	Toronto
8	80	Sales	2500	Oxford

# Creating Joins with the USING Clause

- If several columns have the same names but the data types do not match, use the `USING` clause to specify the columns for the equijoin.
- Use the `USING` clause to match only one column when more than one column matches.
- The `NATURAL JOIN` and `USING` clauses are mutually exclusive.



# Joining Column Names

**EMPLOYEES**

EMPLOYEE_ID	DEPARTMENT_ID
100	90
101	90
102	90
103	60
104	60
107	60
124	50
141	50
142	50
143	50
144	50
149	80
174	80
176	80

...

**DEPARTMENTS**

	DEPARTMENT_ID	DEPARTMENT_NAME
1	10	Administration
2	20	Marketing
3	50	Shipping
4	60	IT
5	80	Sales
6	90	Executive
7	110	Accounting
8	190	Contracting

**Primary key**

**Foreign key**

# Retrieving Records with the USING Clause

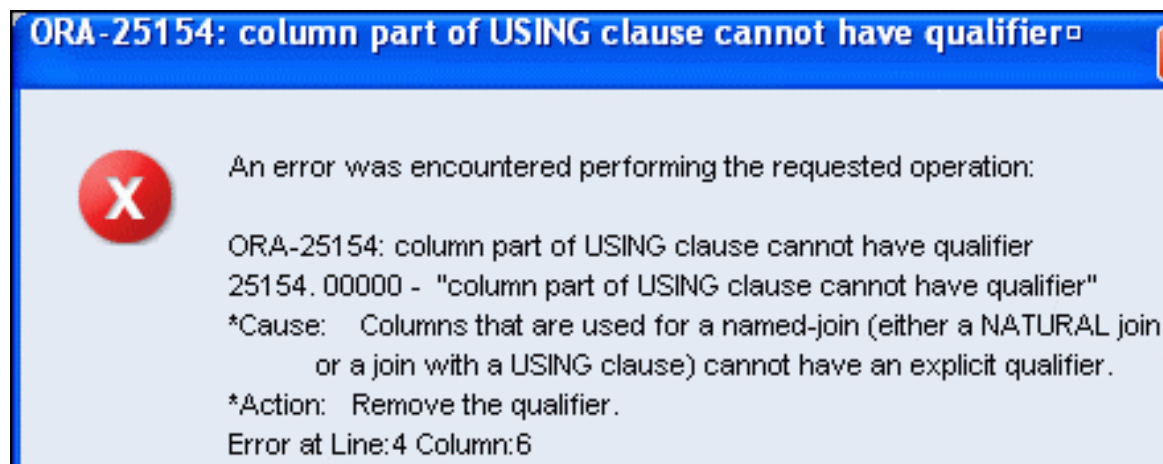
```
SELECT employee_id, last_name,  
       location_id, department_id  
FROM   employees JOIN departments  
USING (department_id) ;
```

	EMPLOYEE_ID	LAST_NAME	LOCATION_ID	DEPARTMENT_ID
1	200	Whalen	1700	10
2	201	Hartstein	1800	20
3	202	Fay	1800	20
4	124	Mourgos	1500	50
5	144	Vargas	1500	50
6	143	Matos	1500	50
7	142	Davies	1500	50
8	141	Rajs	1500	50
9	107	Lorentz	1400	60
10	104	Ernst	1400	60
...				
19	205	Higgins	1700	110

# Using Table Aliases with the USING Clause

- Do not qualify a column that is used in the USING clause.
- If the same column is used elsewhere in the SQL statement, do not alias it.

```
SELECT l.city, d.department_name  
FROM   locations l JOIN departments d  
USING (location_id)  
WHERE d.location_id = 1400;
```



# Creating Joins with the ON Clause

- The join condition for the natural join is basically an equijoin of all columns with the same name.
- Use the ON clause to specify arbitrary conditions or specify columns to join.
- The join condition is separated from other search conditions.
- The ON clause makes code easy to understand.

# Retrieving Records with the ON Clause

```
SELECT e.employee_id, e.last_name, e.department_id,  
       d.department_id, d.location_id  
FROM   employees e JOIN departments d  
ON     (e.department_id = d.department_id);
```

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID_1	LOCATION_ID
1	200	Whalen	10	10	1700
2	201	Hartstein	20	20	1800
3	202	Fay	20	20	1800
4	124	Mourgos	50	50	1500
5	144	Vargas	50	50	1500
6	143	Matos	50	50	1500
7	142	Davies	50	50	1500
8	141	Rajs	50	50	1500
9	107	Lorentz	60	60	1400
10	104	Ernst	60	60	1400

...

# Creating Three-Way Joins with the ON Clause

```
SELECT employee_id, city, department_name
FROM   employees e
JOIN   departments d
ON     d.department_id = e.department_id
JOIN   locations l
ON     d.location_id = l.location_id;
```

	EMPLOYEE_ID	CITY	DEPARTMENT_NAME
1	100	Seattle	Executive
2	101	Seattle	Executive
3	102	Seattle	Executive
4	103	Southlake	IT
5	104	Southlake	IT
6	107	Southlake	IT
7	124	South San Francisco	Shipping
8	141	South San Francisco	Shipping

...

# Applying Additional Conditions to a Join

Use the `AND` clause or the `WHERE` clause to apply additional conditions:

```
SELECT e.employee_id, e.last_name, e.department_id,  
       d.department_id, d.location_id  
FROM   employees e JOIN departments d  
ON     (e.department_id = d.department_id)  
AND    e.manager_id = 149 ;
```

Or

```
SELECT e.employee_id, e.last_name, e.department_id,  
       d.department_id, d.location_id  
FROM   employees e JOIN departments d  
ON     (e.department_id = d.department_id)  
WHERE  e.manager_id = 149 ;
```

# Lesson Agenda

- Types of JOINS and its syntax
- Natural join:
  - USING clause
  - ON clause
- **Self-join**
- Nonequijoins
- OUTER join:
  - LEFT OUTER join
  - RIGHT OUTER join
  - FULL OUTER join
- Cartesian product
  - Cross join



# Joining a Table to Itself

**EMPLOYEES (WORKER)**

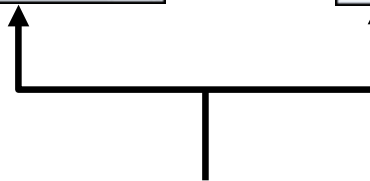
	EMPLOYEE_ID	LAST_NAME	MANAGER_ID
1	100	King	(null)
2	101	Kochhar	100
3	102	De Haan	100
4	103	Hunold	102
5	104	Ernst	103
6	107	Lorentz	103
7	124	Mourgos	100
8	141	Rajs	124
9	142	Davies	124
10	143	Matos	124

...

**EMPLOYEES (MANAGER)**

EMPLOYEE_ID	LAST_NAME
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst
107	Lorentz
124	Mourgos
141	Rajs
142	Davies
143	Matos



...



**MANAGER\_ID in the WORKER table is equal to  
EMPLOYEE\_ID in the MANAGER table.**

# Self-Joins Using the ON Clause

```
SELECT worker.last_name emp, manager.last_name mgr
FROM   employees worker JOIN employees manager
ON     (worker.manager_id = manager.employee_id);
```

	 EMP	 MGR
1	Hunold	De Haan
2	Fay	Hartstein
3	Gietz	Higgins
4	Lorentz	Hunold
5	Ernst	Hunold
6	Zlotkey	King
7	Mourgos	King
8	Kochhar	King
9	Hartstein	King
10	De Haan	King

...

# Lesson Agenda

- Types of JOINS and its syntax
- Natural join:
  - USING clause
  - ON clause
- Self-join
- **Nonequijoins**
- OUTER join:
  - LEFT OUTER join
  - RIGHT OUTER join
  - FULL OUTER join
- Cartesian product
  - Cross join

# Nonequijoins

**EMPLOYEES**

	LAST_NAME	SALARY
1	King	24000
2	Kochhar	17000
3	De Haan	17000
4	Hunold	9000
5	Ernst	6000
6	Lorentz	4200
7	Mourgos	5800
8	Rajs	3500
9	Davies	3100
10	Matos	2600
...		
19	Higgins	12000
20	Gietz	8300

**JOB\_GRADES**

	GRADE_LEVEL	LOWEST_SAL	HIGHEST_SAL
1	A	1000	2999
2	B	3000	5999
3	C	6000	9999
4	D	10000	14999
5	E	15000	24999
6	F	25000	40000

**JOB\_GRADES table defines the LOWEST\_SAL and HIGHEST\_SAL range of values for each GRADE\_LEVEL. Hence, the GRADE\_LEVEL column can be used to assign grades to each employee.**

# Retrieving Records with Nonequijoins

```
SELECT e.last_name, e.salary, j.grade_level  
FROM   employees e JOIN job_grades j  
ON     e.salary  
       BETWEEN j.lowest_sal AND j.highest_sal;
```

	LAST_NAME	SALARY	GRADE_LEVEL
1	Vargas	2500	A
2	Matos	2600	A
3	Davies	3100	B
4	Rajs	3500	B
5	Lorentz	4200	B
6	Whalen	4400	B
7	Mourgos	5800	B
8	Ernst	6000	C
9	Fay	6000	C
10	Grant	7000	C

...

# Lesson Agenda

- Types of JOINS and its syntax
- Natural join:
  - USING clause
  - ON clause
- Self-join
- Nonequijoins
- OUTER join:
  - LEFT OUTER join
  - RIGHT OUTER join
  - FULL OUTER join
- Cartesian product
  - Cross join

# Returning Records with No Direct Match Using OUTER Joins

**DEPARTMENTS**

DEPARTMENT_NAME	DEPARTMENT_ID
Administration	10
Marketing	20
Shipping	50
IT	60
Sales	80
Executive	90
Accounting	110
Contracting	190

There are no employees in department 190.

Employee "Grant" has not been assigned a department ID.

**Equijoin with EMPLOYEES**

	DEPARTMENT_ID	LAST_NAME
1	90	King
2	90	Kochhar
3	90	De Haan
4	60	Hunold
5	60	Ernst
6	60	Lorentz
7	50	Mourgos
8	50	Rajs
9	50	Davies
10	50	Matos

...

18	110	Higgins
19	110	Gietz

# INNER Versus OUTER Joins

- In SQL:1999, the join of two tables returning only matched rows is called an `INNER` join.
- A join between two tables that returns the results of the `INNER` join as well as the unmatched rows from the left (or right) table is called a left (or right) `OUTER` join.
- A join between two tables that returns the results of an `INNER` join as well as the results of a left and right join is a `full OUTER` join.



# LEFT OUTER JOIN

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e LEFT OUTER JOIN departments d
ON     (e.department_id = d.department_id) ;
```

	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	Whalen	10	Administration
2	Fay	20	Marketing
3	Hartstein	20	Marketing
4	Vargas	50	Shipping
5	Matos	50	Shipping

...

17	King	90	Executive
18	Gietz	110	Accounting
19	Higgins	110	Accounting
20	Grant	(null)	(null)

# RIGHT OUTER JOIN

```
SELECT e.last_name, d.department_id, d.department_name
FROM   employees e RIGHT OUTER JOIN departments d
ON     (e.department_id = d.department_id) ;
```

	A 2	LAST_NAME	A 2	DEPARTMENT_ID	A 2	DEPARTMENT_NAME
1		Whalen		10		Administration
2		Hartstein		20		Marketing
3		Fay		20		Marketing
4		Mourgos		50		Shipping

...

18		Gietz		110		Accounting
19		Higgins		110		Accounting
20		(null)		190		Contracting

# FULL OUTER JOIN

```
SELECT e.last_name, d.department_id, d.department_name
FROM   employees e FULL OUTER JOIN departments d
ON     (e.department_id = d.department_id) ;
```

	<small>R2</small> LAST_NAME	<small>R2</small> DEPARTMENT_ID	<small>R2</small> DEPARTMENT_NAME
1	King	90	Executive
2	Kochhar	90	Executive
3	De Haan	90	Executive
4	Hunold	60	IT

...

15	Grant	(null)	(null)
16	Whalen	10	Administration
17	Hartstein	20	Marketing
18	Fay	20	Marketing
19	Higgins	110	Accounting
20	Gietz	110	Accounting
21	(null)	190	Contracting

# Lesson Agenda

- Types of JOINS and its syntax
- Natural join:
  - USING clause
  - ON clause
- Self-join
- Nonequijoin
- OUTER join:
  - LEFT OUTER join
  - RIGHT OUTER join
  - FULL OUTER join
- Cartesian product
  - Cross join

# Cartesian Products

- A Cartesian product is formed when:
  - A join condition is omitted
  - A join condition is invalid
  - All rows in the first table are joined to all rows in the second table
- To avoid a Cartesian product, always include a valid join condition.

# Generating a Cartesian Product

**EMPLOYEES (20 rows)**

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	100	King	90
2	101	Kochhar	90
3	102	De Haan	90
4	103	Hunold	60

...

19	205	Higgins	110
20	206	Gietz	110

**DEPARTMENTS (8 rows)**

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
1	10	Administration	1700
2	20	Marketing	1800
3	50	Shipping	1500
4	60	IT	1400
5	80	Sales	2500
6	90	Executive	1700
7	110	Accounting	1700
8	190	Contracting	1700

**Cartesian product:  
20 x 8 = 160 rows**

	EMPLOYEE_ID	DEPARTMENT_ID	LOCATION_ID
1	100	90	1700
2	101	90	1700
3	102	90	1700
4	103	60	1700

...

159	205	110	1700
160	206	110	1700

# Creating Cross Joins

- The CROSS JOIN clause produces the cross-product of two tables.
- This is also called a Cartesian product between the two tables.

```
SELECT last_name, department_name  
FROM   employees  
CROSS JOIN departments ;
```

	LAST_NAME	DEPARTMENT_NAME
1	Abel	Administration
2	Davies	Administration
3	De Haan	Administration
4	Ernst	Administration
5	Fay	Administration
...		
159	Whalen	Contracting
160	Zlotkey	Contracting

# Quiz

The SQL:1999 standard join syntax supports the following types of joins. Which of these join types does Oracle join syntax support?

1. Equijoins
2. Nonequijoins
3. Left OUTER join
4. Right OUTER join
5. Full OUTER join
6. Self joins
7. Natural joins
8. Cartesian products



# Summary

In this lesson, you should have learned how to use joins to display data from multiple tables by using:

- Equijoins
- Nonequijoins
- OUTER joins
- Self-joins
- Cross joins
- Natural joins
- Full (or two-sided) OUTER joins

# Practice 6: Overview

This practice covers the following topics:

- Joining tables using an equijoin
- Performing outer and self-joins
- Adding conditions



# Using Subqueries to Solve Queries

# Objectives

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

- Define subqueries
- Describe the types of problems that the subqueries can solve
- List the types of subqueries
- Write single-row and multiple-row subqueries

# Lesson Agenda

- Subquery: Types, syntax, and guidelines
- Single-row subqueries:
  - Group functions in a subquery
  - HAVING clause with subqueries
- Multiple-row subqueries
  - Use ALL or ANY operator.
- Null values in a subquery

# Using a Subquery to Solve a Problem

Who has a salary greater than Abel's?

**Main query:**



**Which employees have salaries greater than Abel's salary?**

**Subquery:**



**What is Abel's salary?**



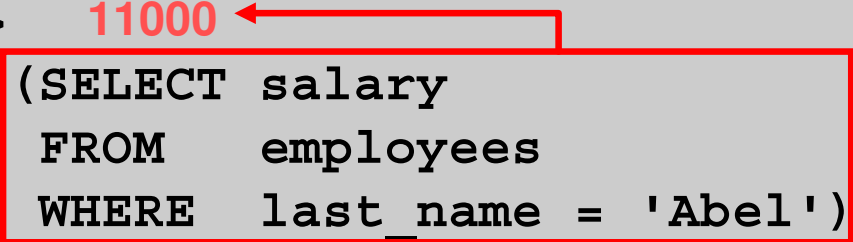
# Subquery Syntax

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

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

# Using a Subquery

```
SELECT last_name, salary
FROM employees
WHERE salary > 11000
      (SELECT salary
       FROM employees
       WHERE last_name = 'Abel');
```



	LAST_NAME	SALARY
1	King	24000
2	Kochhar	17000
3	De Haan	17000
4	Hartstein	13000
5	Higgins	12000



# Guidelines for Using Subqueries

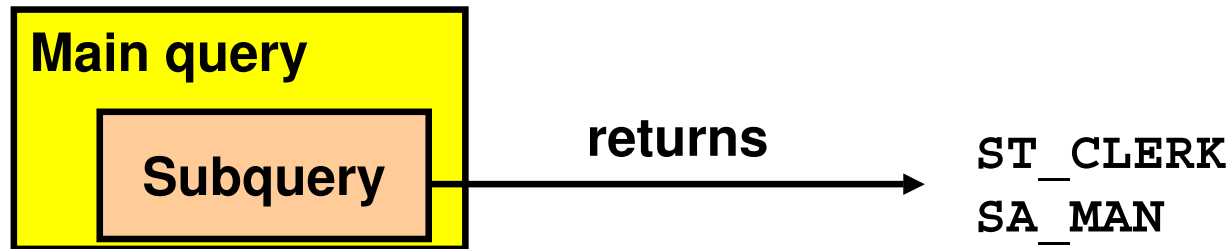
- Enclose subqueries in parentheses.
- Place subqueries on the right side of the comparison condition for readability (However, the subquery can appear on either side of the comparison operator.).
- Use single-row operators with single-row subqueries and multiple-row operators with multiple-row subqueries.

# Types of Subqueries

- Single-row subquery



- Multiple-row subquery



# Lesson Agenda



- Subquery: Types, syntax, and guidelines
- Single-row subqueries:
  - Group functions in a subquery
  - HAVING clause with subqueries
- Multiple-row subqueries
  - Use ALL or ANY operator
- Null values in a subquery




# Single-Row Subqueries

- Return only one row
- Use single-row comparison operators


Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to




# Executing Single-Row Subqueries

```
SELECT last_name, job_id, salary
FROM employees
WHERE job_id =  (SELECT job_id
FROM employees
WHERE last_name = 'Taylor')
AND salary >  (SELECT salary
FROM employees
WHERE last_name = 'Taylor');
```

	 LAST_NAME	 JOB_ID	 SALARY
1	Abel	SA_REP	11000

# Using Group Functions in a Subquery

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

	 LAST_NAME	 JOB_ID	 SALARY
1	Vargas	ST_CLERK	2500

# The HAVING Clause with Subqueries

- The Oracle server executes the subqueries first.
- The Oracle server returns results into the HAVING clause of the main query.

```
SELECT    department_id, MIN(salary)
FROM      employees
GROUP BY  department_id
HAVING    MIN(salary) > (SELECT MIN(salary)
                        FROM      employees
                        WHERE     department_id = 50);
```

2500

	DEPARTMENT_ID	MIN(SALARY)
1	(null)	7000
2	90	17000
3	20	6000
...		
7	10	4400

# What Is Wrong with This Statement?

```
SELECT employee_id, last_name
FROM employees
WHERE salary =
      (SELECT MIN(salary)
       FROM employees
       GROUP BY department_id);
```

ORA-01427: single-row subquery returns more than one ...



An error was encountered performing the requested operation:

ORA-01427: single-row subquery returns more than one row  
01427. 00000 - "single-row subquery returns more than one row"

\*Cause:

\*Action:

Error at Line:1

**Single-row operator  
with multiple-row  
subquery**



# No Rows Returned by the Inner Query

```
SELECT last_name, job_id
FROM   employees
WHERE  job_id =
      (SELECT job_id
       FROM   employees
       WHERE  last_name = 'Haas');
```

0 rows selected

**Subquery returns no rows because there is no employee named “Haas.”**

# Lesson Agenda

- Subquery: Types, syntax, and guidelines
- Single-row subqueries:
  - Group functions in a subquery
  - `HAVING` clause with subqueries
- **Multiple-row subqueries**
  - Use `ALL` or `ANY` operator
- Null values in a subquery

# Multiple-Row Subqueries

- Return more than one row
- Use multiple-row comparison operators

Operator	Meaning
IN	Equal to any member in the list
ANY	Must be preceded by =, !=, >, <, <=, >=. Compares a value to each value in a list or returned by a query. Evaluates to <code>FALSE</code> if the query returns no rows.
ALL	Must be preceded by =, !=, >, <, <=, >=. Compares a value to every value in a list or returned by a query. Evaluates to <code>TRUE</code> if the query returns no rows.

# Using the ANY Operator in Multiple-Row Subqueries

```
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE salary < ANY
      (SELECT salary
       FROM employees
       WHERE job_id = 'IT_PROG')
AND job_id <> 'IT_PROG';
```

9000, 6000, 4200

	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	144	Vargas	ST_CLERK	2500
2	143	Matos	ST_CLERK	2600
3	142	Davies	ST_CLERK	3100
4	141	Rajs	ST_CLERK	3500
5	200	Whalen	AD_ASST	4400

...

9	206	Gietz	AC_ACCOUNT	8300
10	176	Taylor	SA_REP	8600

# Using the ALL Operator in Multiple-Row Subqueries

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary < ALL
      (SELECT salary
       FROM   employees
       WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';
```

9000, 6000, 4200

	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	141	Rajs	ST_CLERK	3500
2	142	Davies	ST_CLERK	3100
3	143	Matos	ST_CLERK	2600
4	144	Vargas	ST_CLERK	2500

# Lesson Agenda

- Subquery: Types, syntax, and guidelines
- Single-row subqueries:
  - Group functions in a subquery
  - HAVING clause with subqueries
- Multiple-row subqueries
  - Use ALL or ANY operator
- Null values in a subquery

# Null Values in a Subquery

```
SELECT emp.last_name  
FROM   employees emp  
WHERE  emp.employee_id NOT IN  
                                (SELECT mgr.manager_id  
                                FROM   employees mgr);
```

0 rows selected

# Quiz

Using a subquery is equivalent to performing two sequential queries and using the result of the first query as the search value(s) in the second query.

1. True
2. False



# Summary

In this lesson, you should have learned how to:

- Identify when a subquery can help solve a problem
- Write subqueries when a query is based on unknown values

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

# Practice 7: Overview

This practice covers the following topics:

- Creating subqueries to query values based on unknown criteria
- Using subqueries to find out the values that exist in one set of data and not in another

# 8

## Using the Set Operators

# Objectives

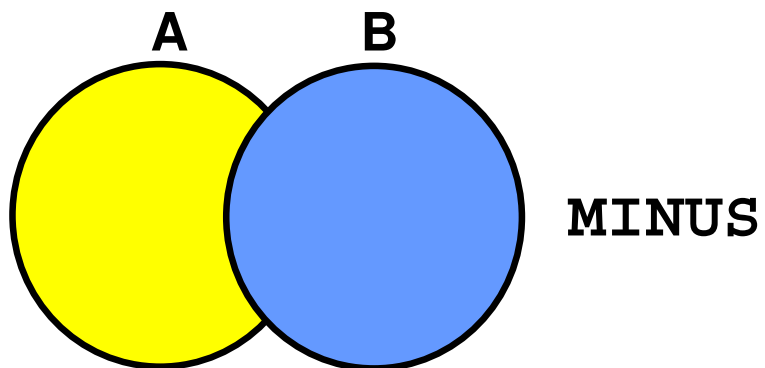
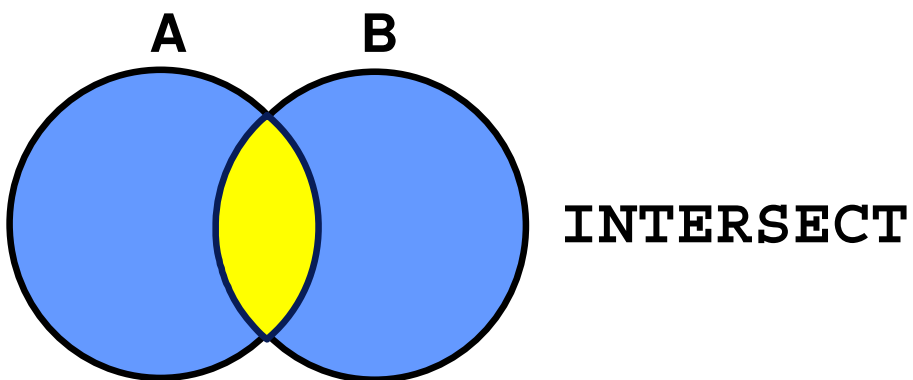
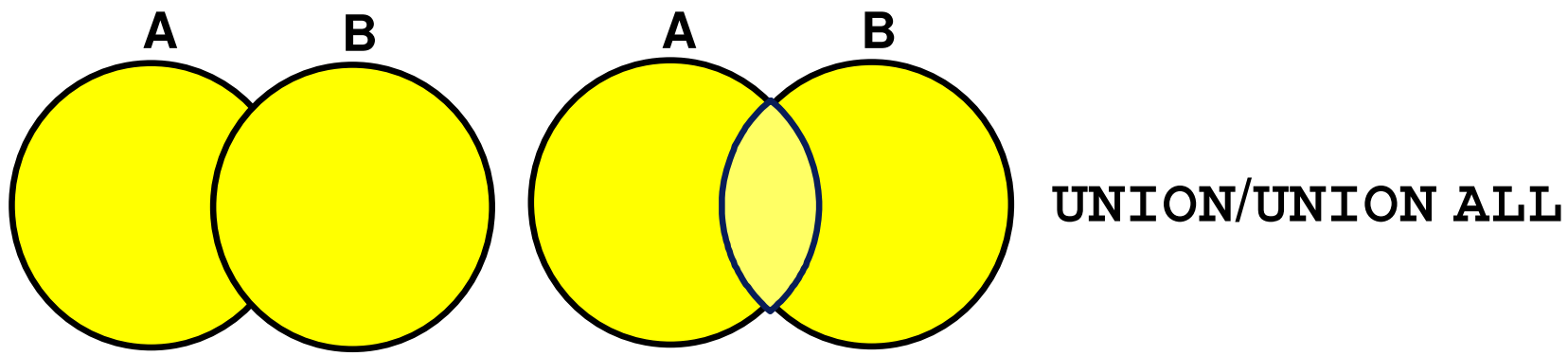
After completing this lesson, you should be able to do the following:

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

# Lesson Agenda

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

# Set Operators



# Set Operator Guidelines

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

# The 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`.



# Lesson Agenda

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

# Tables Used in This Lesson

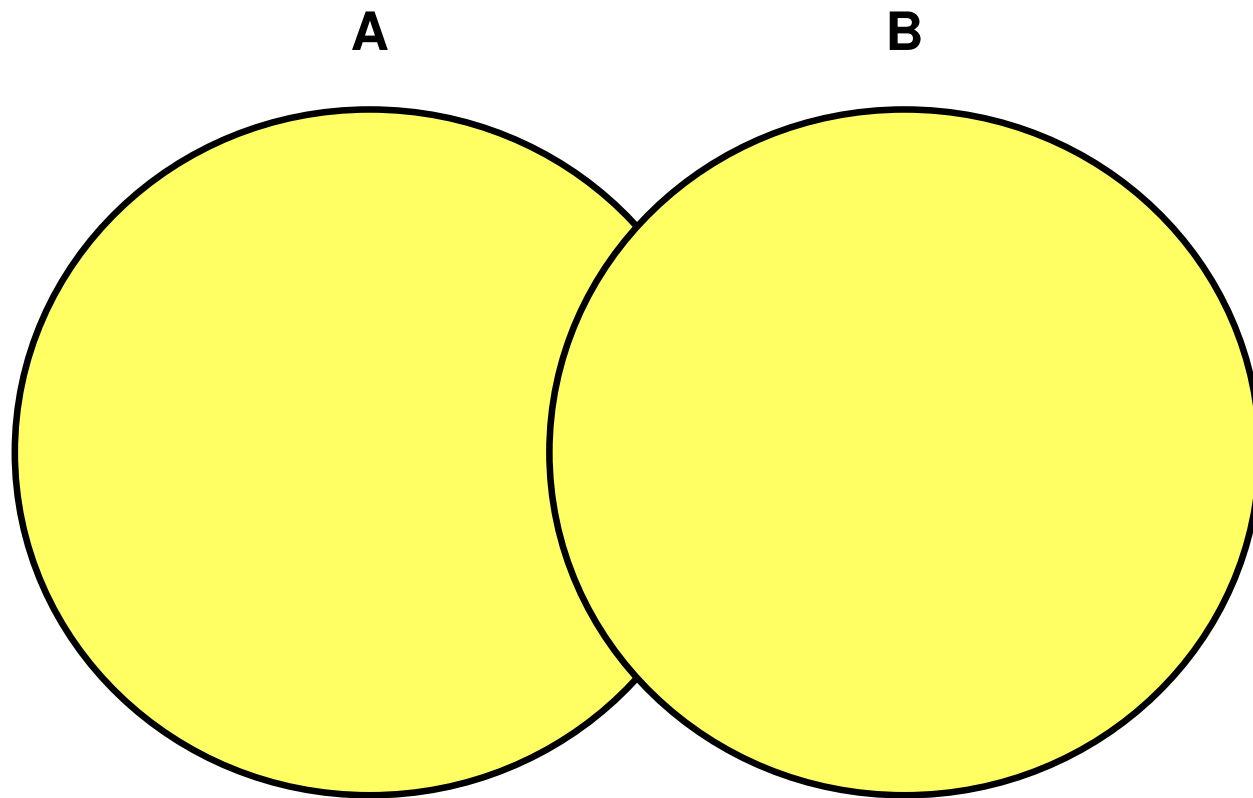
The tables used in this lesson are:

- `EMPLOYEES`: Provides details regarding all current employees
- `JOB_HISTORY`: Records the details of the start date and end date of the former job, and the job identification number and department when an employee switches jobs

# Lesson Agenda

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

# UNION Operator



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

# Using the UNION Operator

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

```
SELECT employee_id, job_id
FROM employees
UNION
SELECT employee_id, job_id
FROM job_history;
```

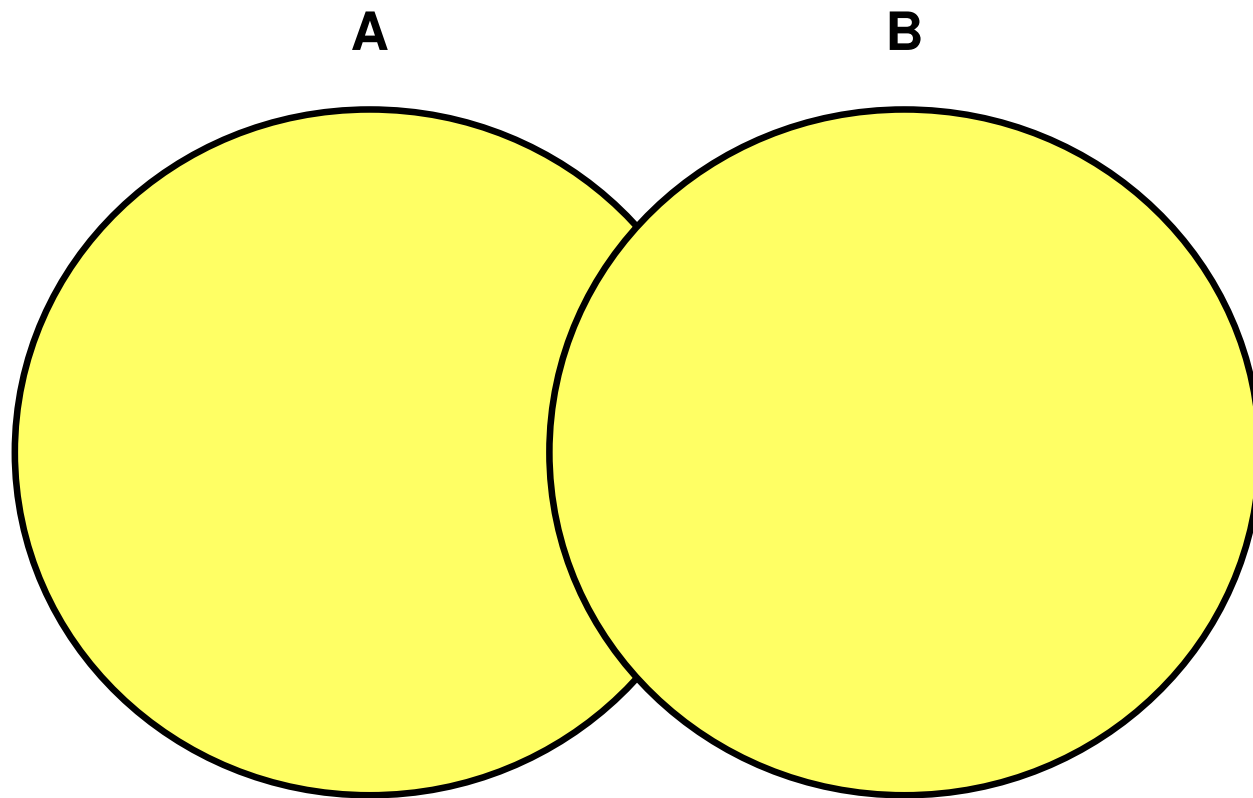
	EMPLOYEE_ID	JOB_ID
1	100	AD_PRES
2	101	AC_ACCOUNT

...

22	200	AC_ACCOUNT
23	200	AD_ASST
24	201	MK_MAN

...

# UNION ALL Operator



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

# Using the UNION ALL Operator

Display the current and previous departments of all employees.

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

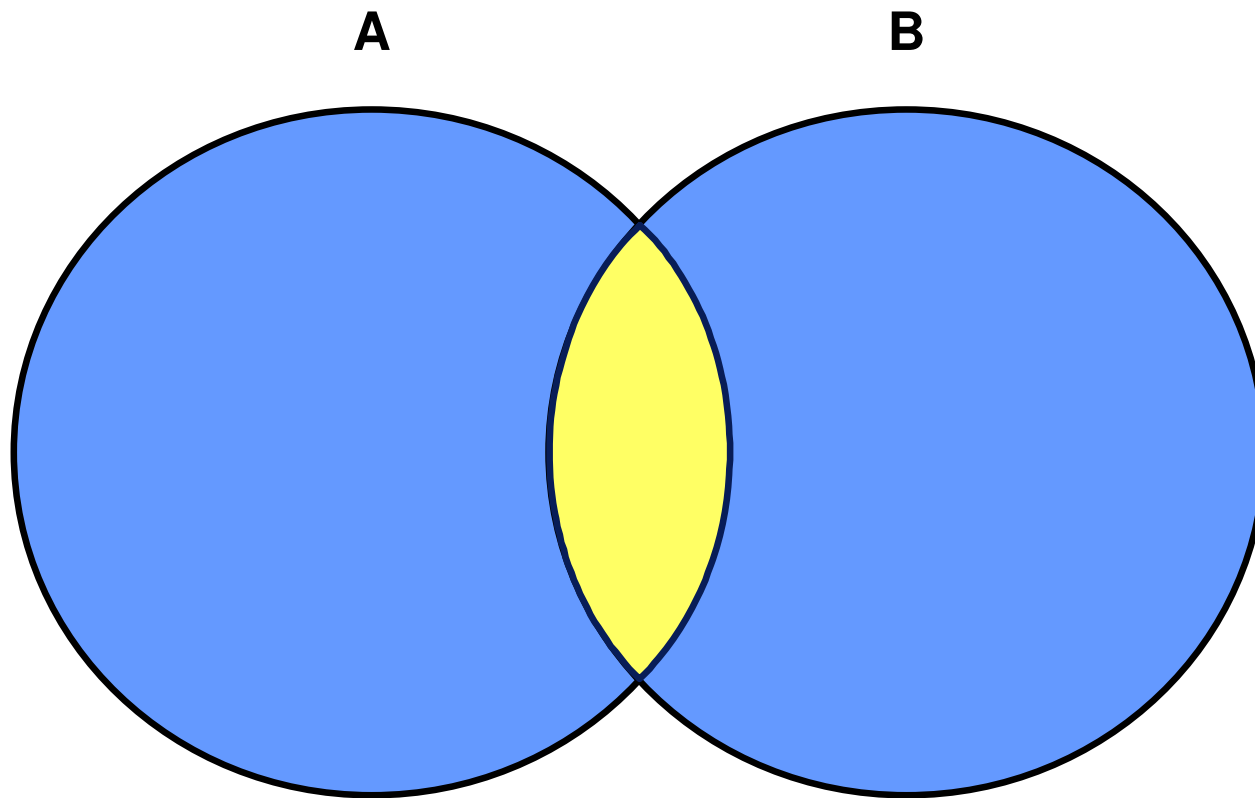
	EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
1	100	AD_PRES	90
...			
16	144	ST_CLERK	50
17	149	SA_MAN	80
18	174	SA_REP	80
19	176	SA_REP	80
20	176	SA_MAN	80
21	176	SA_REP	80
22	178	SA_REP	(null)
...			
30	206	AC_ACCOUNT	110

# Lesson Agenda

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



# INTERSECT Operator



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

# Using the INTERSECT Operator

Display the employee IDs and job IDs of those employees who currently have a job title that is the same as their previous one (that is, they changed jobs but have now gone back to doing the same job they did previously).

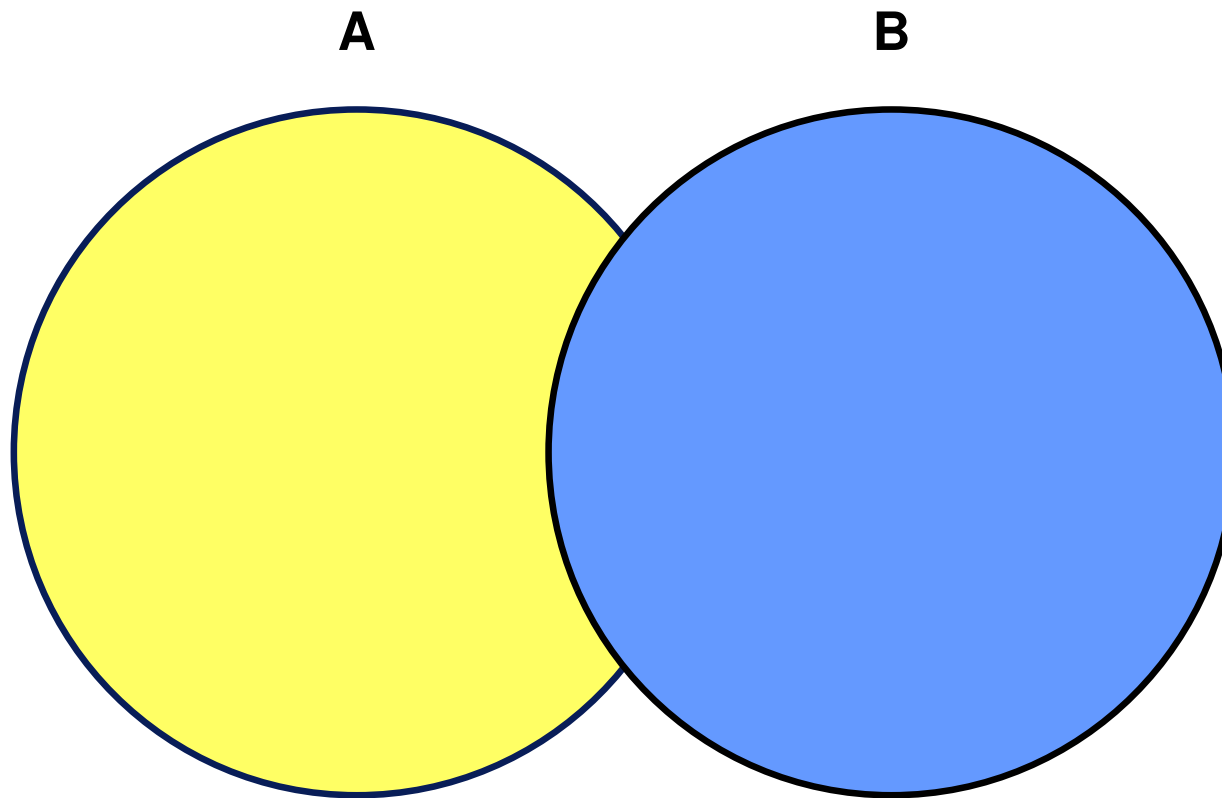
```
SELECT employee_id, job_id
FROM   employees
INTERSECT
SELECT employee_id, job_id
FROM   job_history;
```

	EMPLOYEE_ID	JOB_ID
1	176	SA_REP
2	200	AD_ASST

# Lesson Agenda

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

# MINUS Operator



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

# Using the MINUS Operator

Display the employee IDs of those employees who have not changed their jobs even once.

```
SELECT employee_id
FROM   employees
MINUS
SELECT employee_id
FROM   job_history;
```

	EMPLOYEE_ID
1	100
2	103
3	104
4	107
5	124
...	
14	205
15	206

# Lesson Agenda

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

# Matching the SELECT Statements

- Using the UNION operator, display the location ID, department name, and the state where it is located.
- 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;
```

# Matching the SELECT Statement: Example

Using the UNION operator, display the employee ID, job ID, and salary of all employees.

```
SELECT employee_id, job_id, salary
FROM   employees
UNION
SELECT employee_id, job_id, 0
FROM   job_history;
```

	EMPLOYEE_ID	JOB_ID	SALARY
1	100	AD_PRES	24000
2	101	AC_ACCOUNT	0
3	101	AC_MGR	0
4	101	AD_VP	17000
5	102	AD_VP	17000
...			
29	205	AC_MGR	12000
30	206	AC_ACCOUNT	8300



# Lesson Agenda

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

# Using the ORDER BY Clause in Set Operations

- The ORDER BY clause can appear only once at the end of the compound query.
- Component queries cannot have individual ORDER BY clauses.
- 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 an ascending order.

# Quiz

Identify the set operator guidelines.

1. The expressions in the `SELECT` lists must match in number.
2. Parentheses may not be used to alter the sequence of execution.
3. The data type of each column in the second query must match the data type of its corresponding column in the first query.
4. The `ORDER BY` clause can be used only once in a compound query, unless a `UNION ALL` operator is used.

# 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

# Practice 8: Overview

In this practice, you create reports by using:

- The UNION operator
- The INTERSECTION operator
- The MINUS operator



# **Manipulating Data**

# Objectives

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

- Describe each data manipulation language (DML) statement
- Insert rows into a table
- Update rows in a table
- Delete rows from a table
- Control transactions

# 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 transactions control using COMMIT, ROLLBACK, and SAVEPOINT
- Read consistency
- FOR UPDATE clause in a SELECT statement



# 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.

# Adding a New Row to 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

70	Public Relations	100	1700
----	------------------	-----	------

**New  
row**

**Insert new row  
into 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
8	190	Contracting	(null)	1700

9	70	Public Relations	100	1700
---	----	------------------	-----	------

# 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.

# 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 rows inserted
```

- Enclose character and date 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 rows inserted

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

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

1 rows inserted

# Inserting Special Values

The SYSDATE function records the current date and time.

```
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',  
       SYSDATE, 'AC_ACCOUNT', 6900,  
       NULL, 205, 110);
```

```
1 rows inserted
```

# 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, 1999', 'MON DD, YYYY'),
             'SA_REP', 11000, 0.2, 100, 60);
```

1 rows inserted

- Verify your addition.

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT
114	Den	Raphealy	DRAPHEAL	515.127.4561	03-FEB-99	SA_REP	11000	0.2

# Creating a Script

- Use & 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_id);
```

The image shows three overlapping dialog boxes titled "Enter Substitution Variable". The first dialog box in the background is for "DEPARTMENT\_ID:" with the value "40" entered and an "OK" button. The second dialog box in the middle is for "DEPARTMENT\_NAME:" with the value "Human Resources" entered and an "OK" button. The third dialog box in the foreground is for "LOCATION:" with the value "2500" entered and both "OK" and "Cancel" buttons.



# 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%';
```

4 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.

# 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 transactions control using COMMIT, ROLLBACK, and SAVEPOINT
- Read consistency
- FOR UPDATE clause in a SELECT statement

# 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

# 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).

# 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 rows 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.

# Updating Two Columns with a Subquery

Update employee 113's job and salary to match those of employee 205.

```
UPDATE    employees
SET       job_id    = (SELECT  job_id
                        FROM      employees
                        WHERE     employee_id = 205),
          salary    = (SELECT  salary
                        FROM      employees
                        WHERE     employee_id = 205)
WHERE     employee_id    = 113;
```

```
1 rows updated
```

# 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 rows updated
```

# 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 transactions control using COMMIT, ROLLBACK, and SAVEPOINT
- Read consistency
- FOR UPDATE clause in a SELECT statement



# 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

# DELETE Statement

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

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

# Deleting Rows from a Table

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

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

```
1 rows deleted
```

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

```
DELETE FROM copy_emp;
```

```
22 rows deleted
```

# 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 =
    (SELECT department_id
     FROM departments
     WHERE department_name
           LIKE '%Public%');
```

1 rows deleted

# 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 easily be undone
- Syntax:

```
TRUNCATE TABLE table_name;
```

- Example:

```
TRUNCATE TABLE copy_emp;
```

# 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 transactions control using COMMIT, ROLLBACK, and SAVEPOINT**
- Read consistency
- FOR UPDATE clause in a SELECT statement

# 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

# 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.

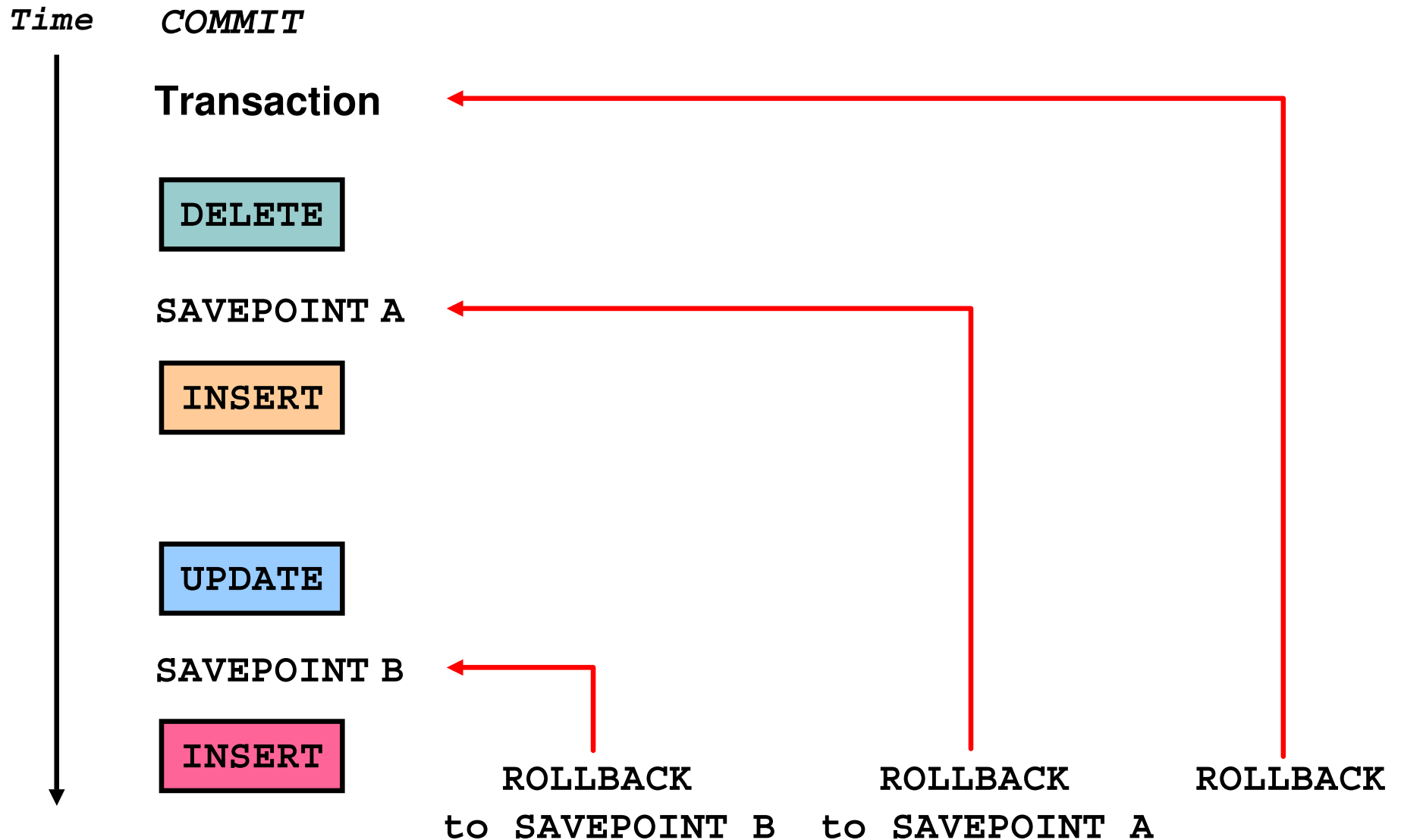


# Advantages of COMMIT and ROLLBACK Statements

With COMMIT and ROLLBACK statements, you can:

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

# Explicit Transaction Control Statements



# 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.

```
UPDATE...
```

```
SAVEPOINT update_done;
```

```
SAVEPOINT update_done succeeded.
```

```
INSERT...
```

```
ROLLBACK TO update_done;
```

```
ROLLBACK TO succeeded.
```

# Implicit Transaction Processing

- An automatic commit occurs in the following circumstances:
  - A DDL statement is issued
  - A DCL statement is issued
  - 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.

# State of the Data

## Before COMMIT or ROLLBACK

- The previous state of the data can be recovered.
- The current user can review the results of the DML operations by using the `SELECT` statement.
- Other users *cannot* view the results of the DML statements issued by the current user.
- The affected rows are *locked*; other users cannot change the data in the affected rows.

# State of the Data After COMMIT

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

# Committing Data

- Make the changes:

```
DELETE FROM employees  
WHERE employee_id = 99999;
```

```
1 rows deleted
```

```
INSERT INTO departments  
VALUES (290, 'Corporate Tax', NULL, 1700);
```

```
1 rows inserted
```

- Commit the changes:

```
COMMIT;
```

```
COMMIT succeeded.
```

# State of the 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 ;
```



# State of the Data After ROLLBACK: Example

```
DELETE FROM test;  
25,000 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.
```

# 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.

# 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 transactions control using COMMIT, ROLLBACK, and SAVEPOINT
- Read consistency
- FOR UPDATE clause in a SELECT statement

# Read Consistency

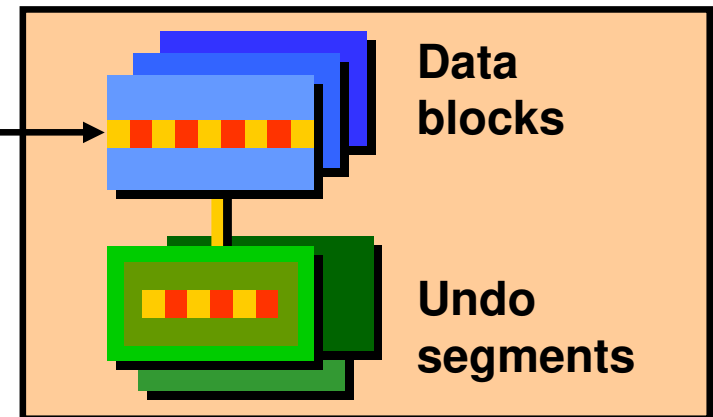
- Read consistency guarantees a consistent view of the data at all times.
- Changes made by one user do not conflict with the changes made by another user.
- Read consistency ensures that, on the same data:
  - Readers do not wait for writers
  - Writers do not wait for readers
  - Writers wait for writers

# Implementing Read Consistency

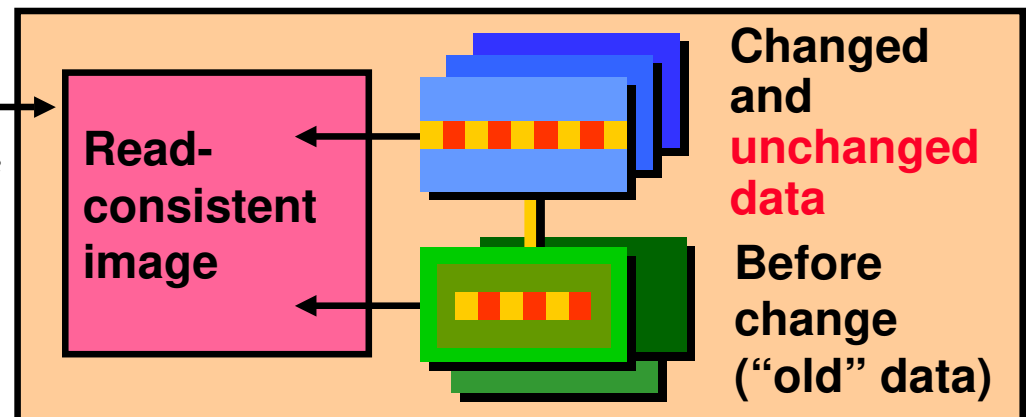
User A



```
UPDATE employees  
SET    salary = 7000  
WHERE  last_name = 'Grant';
```



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



User B

# 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 transactions control using COMMIT, ROLLBACK, and SAVEPOINT
- Read consistency
- **FOR UPDATE clause in a SELECT statement**

# 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, then the database waits until the row is available, and then returns the results of the SELECT statement.

# 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 you intend to change, then only the rows from that specific table are locked.



# Quiz

The following statements produce the same results:

```
DELETE FROM copy_emp;
```

```
TRUNCATE TABLE copy_emp;
```

1. True
2. False

# 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

# Practice 9: Overview

This practice covers the following topics:

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

# 10

## **Using DDL Statements to Create and Manage Tables**

# 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
- Describe how schema objects work

# Lesson Agenda

- Database objects
  - Naming rules
- CREATE TABLE statement:
  - Access another user's tables
  - DEFAULT option
- Data types
- Overview of constraints: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK constraints
- Creating a table using a subquery
- ALTER TABLE
  - Read-only tables
- DROP TABLE statement

# 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 some queries
Synonym	Gives alternative name to an object

# Naming Rules

Table names and column names:

- Must begin with a letter
- Must be 1–30 characters long
- Must contain only A–Z, a–z, 0–9, \_, \$, and #
- Must not duplicate the name of another object owned by the same user
- Must not be an Oracle server–reserved word



# Lesson Agenda

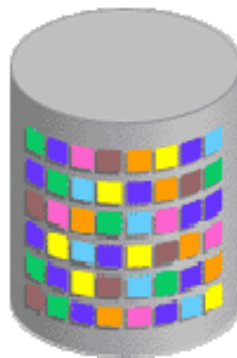
- Database objects
  - Naming rules
- CREATE TABLE statement:
  - Access another user's tables
  - DEFAULT option
- Data types
- Overview of constraints: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK constraints
- Creating a table using a subquery
- ALTER TABLE
  - Read-only tables
- DROP TABLE statement

# CREATE TABLE Statement

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

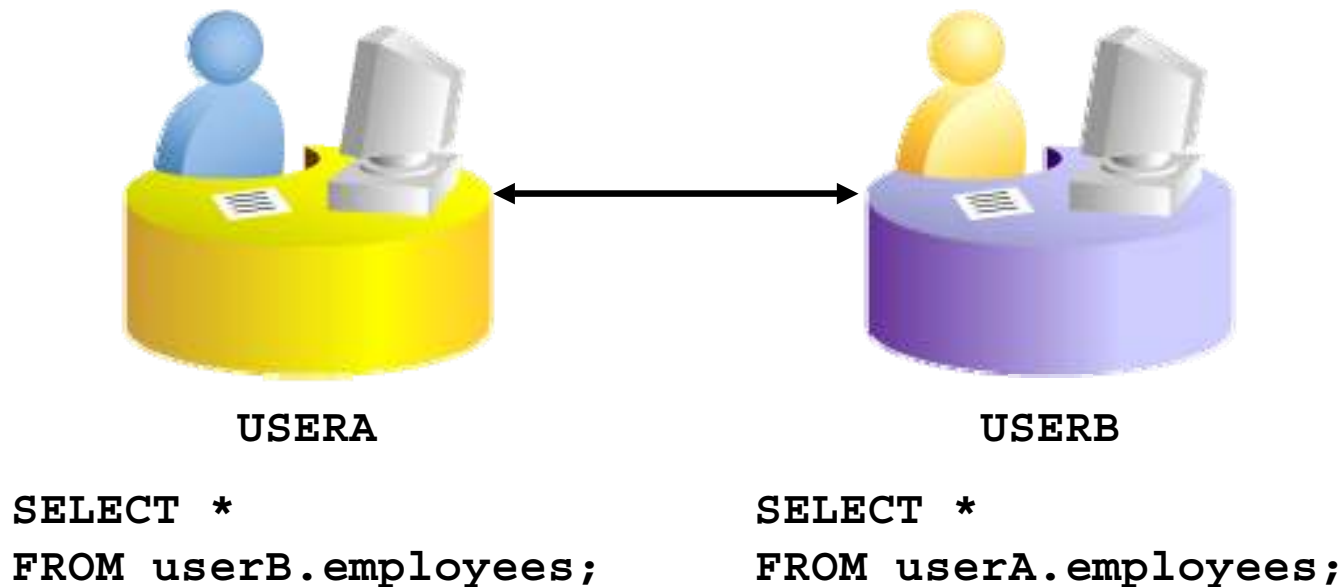
```
CREATE TABLE [schema.] table  
              (column datatype [DEFAULT expr] [, ...]);
```

- You specify:
  - Table name
  - Column name, column data type, and column size



# 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.



# DEFAULT Option

- Specify a default value for a column during an insert.

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

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

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

```
CREATE TABLE succeeded.
```

# Creating Tables

- Create the table:

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

```
CREATE TABLE succeeded.
```

- Confirm table creation:

```
DESCRIBE dept
```

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

# Lesson Agenda

- Database objects
  - Naming rules
- CREATE TABLE statement:
  - Access another user's tables
  - DEFAULT option
- **Data types**
- Overview of constraints: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK constraints
- Creating a table using a subquery
- ALTER TABLE
  - Read-only tables
- DROP TABLE statement

# Data Types

Data Type	Description
VARCHAR2 ( <i>size</i> )	Variable-length character data
CHAR ( <i>size</i> )	Fixed-length character data
NUMBER ( <i>p</i> , <i>s</i> )	Variable-length numeric data
DATE	Date and time values
LONG	Variable-length character data (up to 2 GB)
CLOB	Character data (up to 4 GB)
RAW and LONG RAW	Raw binary data
BLOB	Binary data (up to 4 GB)
BFILE	Binary data stored in an external file (up to 4 GB)
ROWID	A base-64 number system representing the unique address of a row in its table

# Datetime Data Types

You can use several datetime data types:

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





# Lesson Agenda

- Database objects
  - Naming rules
- CREATE TABLE statement:
  - Access another user's tables
  - DEFAULT option
- Data types
- **Overview of constraints: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK constraints**
- Creating a table using a subquery
- ALTER TABLE
  - Read-only tables
- DROP TABLE statement

# Including Constraints

- Constraints enforce rules at the table level.
- Constraints prevent the deletion of a table if there are dependencies.
- The following constraint types are valid:
  - NOT NULL
  - UNIQUE
  - PRIMARY KEY
  - FOREIGN KEY
  - CHECK



# 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 same time as the creation of the table
  - After the creation of the table
- Define a constraint at the column or table level.
- View a constraint in the data dictionary.

# 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, ...),
```

# Defining Constraints

- 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

# NOT NULL Constraint

Ensures that null values are not permitted for the column:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	COMMISSION_PCT
100	Steven	King	SKING	17-JUN-87	AD_PRES	(null)
101	Neena	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	(null)
102	Lex	De Haan	LDEHAAN	13-JAN-93	AD_VP	(null)
103	Alexander	Hunold	AHUNOLD	03-JAN-90	IT_PROG	(null)
104	Bruce	Ernst	BERNST	21-MAY-91	IT_PROG	(null)
107	Diana	Lorentz	DLORENTZ	07-FEB-99	IT_PROG	(null)
124	Kevin	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	(null)
141	Trenna	Rajs	TRAJS	17-OCT-95	ST_CLERK	(null)
142	Curtis	Davies	CDAVIES	29-JAN-97	ST_CLERK	(null)
143	Randall	Matos	RMATOS	15-MAR-98	ST_CLERK	(null)
144	Peter	Vargas	PVARGAS	09-JUL-98	ST_CLERK	(null)
149	Eleni	Zlotkey	EZLOTKEY	29-JAN-00	SA_MAN	0.2
174	Ellen	Abel	EABEL	11-MAY-96	SA_REP	0.3

...


↑  
**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.)**

# UNIQUE Constraint

## EMPLOYEES



EMPLOYEE_ID	LAST_NAME	EMAIL
100	King	SKING
101	Kochhar	NKOCHHAR
102	De Haan	LDEHAAN
103	Hunold	AHUNOLD
104	Ernst	BERNST
107	Lorentz	DLORENTZ

...



INSERT INTO

208	SMITH	JSMITH
209	SMITH	JSMITH

← Allowed

← Not allowed:  
already exists

# UNIQUE Constraint





Defined 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));
```



# PRIMARY KEY Constraint

DEPARTMENTS PRIMARY KEY

	 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

Not allowed  
(null value)

 INSERT INTO

	Public Accounting	124	2500
	50 Finance	124	1500

Not allowed  
(50 already exists)

# FOREIGN KEY Constraint

**DEPARTMENTS**

**PRIMARY KEY** →

	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

...

**EMPLOYEES**

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	
1	100	King	90	← <b>FOREIGN KEY</b>
2	101	Kochhar	90	
3	102	De Haan	90	
4	103	Hunold	60	
5	104	Ernst	60	

...



**INSERT INTO**

200	Ford	9
201	Ford	60

← **Not allowed  
(9 does not  
exist)**

← **Allowed**

# FOREIGN KEY Constraint

Defined 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));
```

# 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

# CHECK Constraint

- Defines a condition that each row must satisfy
- The following expressions are not allowed:
  - References to CURRVAL, NEXTVAL, LEVEL, and ROWNUM pseudocolumns
  - Calls to SYSDATE, UID, USER, and USERENV functions
  - Queries that refer to other values in other rows

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

# CREATE TABLE: Example

```
CREATE TABLE employees
( employee_id      NUMBER(6)
  CONSTRAINT emp_employee_id PRIMARY KEY
, first_name       VARCHAR2(20)
, last_name        VARCHAR2(25)
  CONSTRAINT emp_last_name_nn NOT NULL
, email            VARCHAR2(25)
  CONSTRAINT emp_email_nn    NOT NULL
  CONSTRAINT emp_email_uk    UNIQUE
, phone_number     VARCHAR2(20)
, hire_date        DATE
  CONSTRAINT emp_hire_date_nn NOT NULL
, job_id           VARCHAR2(10)
  CONSTRAINT emp_job_nn      NOT NULL
, salary           NUMBER(8,2)
  CONSTRAINT emp_salary_ck   CHECK (salary>0)
, commission_pct   NUMBER(2,2)
, manager_id       NUMBER(6)
  CONSTRAINT emp_manager_fk REFERENCES
    employees (employee_id)
, department_id    NUMBER(4)
  CONSTRAINT emp_dept_fk     REFERENCES
    departments (department_id));
```

# Violating Constraints

```
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:
SQL Error: ORA-02291: integrity constraint (ORA16.EMP_DEPT_FK) violated - parent key not found
02291. 00000 - "integrity constraint (%s.%s) violated - parent key not found"
*Cause:      A foreign key value has no matching primary key value.
*Action:     Delete the foreign key or add a matching primary key.
```

Department 55 does not exist.

# 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:

SQL Error: ORA-02292: integrity constraint (ORA16.EMP\_DEPT\_FK) violated - child record found

02292. 00000 - "integrity constraint (%s.%s) violated - child record found"

\*Cause: attempted to delete a parent key value that had a foreign  
dependency.

\*Action: delete dependencies first then parent or disable constraint.



# Lesson Agenda

- Database objects
  - Naming rules
- CREATE TABLE statement:
  - Access another user's tables
  - DEFAULT option
- Data types
- Overview of constraints: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK constraints
- **Creating a table using a subquery**
- ALTER TABLE
  - Read-only tables
- DROP TABLE statement

# 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.

# 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;
```

CREATE TABLE succeeded.

```
DESCRIBE dept80
```

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

# Lesson Agenda

- Database objects
  - Naming rules
- CREATE TABLE statement:
  - Access another user's tables
  - DEFAULT option
- Data types
- Overview of constraints: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK constraints
- Creating a table using a subquery
- ALTER TABLE
  - Read-only tables
- DROP TABLE statement

# 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

# Read-Only Tables

You can use the `ALTER TABLE` syntax to:

- Put a table into 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;
```

# Lesson Agenda

- Database objects
  - Naming rules
- CREATE TABLE statement:
  - Access another user's tables
  - DEFAULT option
- Data types
- Overview of constraints: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK constraints
- Creating a table using a subquery
- ALTER TABLE
  - Read-only tables
- DROP TABLE statement

# 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;
```

```
DROP TABLE dept80 succeeded.
```



# Quiz

You can use constraints to do the following:

1. Enforce rules on the data in a table whenever a row is inserted, updated, or deleted.
2. Prevent the deletion of a table.
3. Prevent the creation of a table.
4. Prevent the creation of data in a table.

# Summary

In this lesson, you should have learned how to use the `CREATE TABLE` statement to create a table 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
- Describe how schema objects work

# Practice 10: 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
- Setting a table to read-only status
- Dropping tables

# 11

## Creating Other Schema Objects

# Objectives

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

- Create simple and complex views
- Retrieve data from views
- Create, maintain, and use sequences
- Create and maintain indexes
- Create private and public synonyms

# Lesson Agenda

- Overview of views:
  - Creating, modifying, and retrieving data from a view
  - Data manipulation language (DML) operations on a view
  - Dropping a view
- Overview of sequences:
  - Creating, using, and modifying a sequence
  - Cache sequence values
  - NEXTVAL and CURRVAL pseudocolumns
- Overview of indexes
  - Creating, dropping indexes
- Overview of synonyms
  - Creating, dropping synonyms

# 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

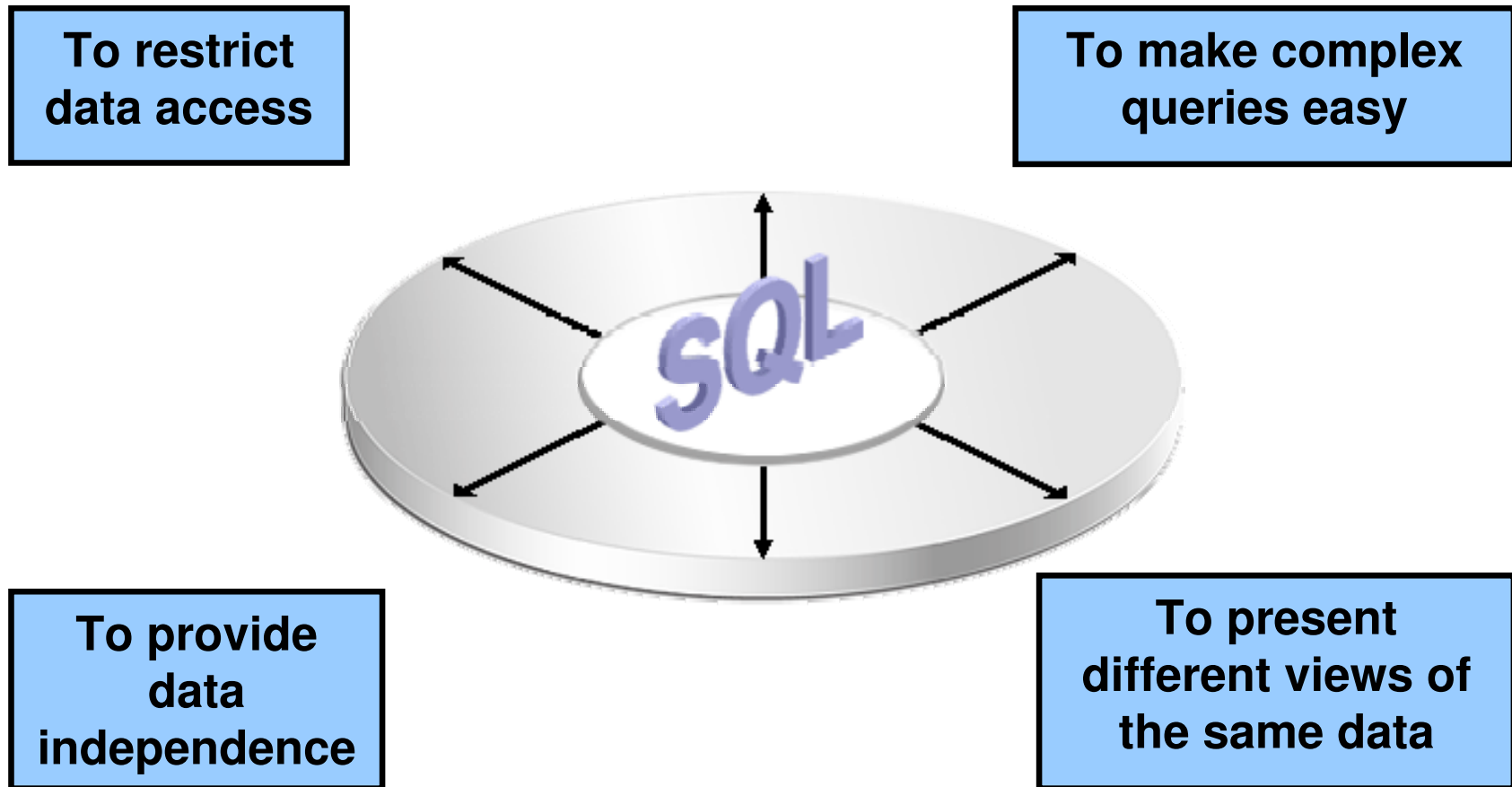
# What Is a View?

## EMPLOYEES table

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
1	100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000
2	101	Neena	Kochhar	NKOCHH...	515.123.4568	21-SEP-89	AD_VP	17000
3	102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000
4	103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000
5							IT_PROG	6000
6							IT_PROG	4200
7							ANALYST	5800
							MARKET	3500
							MARKET	3100
							MARKET	2600
							MARKET	2500
							MANAGER	10500
							SA_REP	11000
							SA_REP	8600
							SA_REP	7000
							AD_ASST	4400
							MK_MAN	13000
							MK_REP	6000
19	205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	12000
20	206	William	Gietz	WGIEZT	515.123.8181	07-JUN-94	AC_ACC...	8300



# Advantages of Views



# 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

# 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.

# 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;
```

```
CREATE VIEW succeeded.
```

- Describe the structure of the view by using the *iSQL\*Plus* DESCRIBE command:

```
DESCRIBE empvu80
```

# 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;
CREATE VIEW succeeded.
```

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

# Retrieving Data from a View

```
SELECT *  
FROM salvu50;
```

	<small>A 2</small>	ID_NUMBER	<small>A 2</small>	NAME	<small>A 2</small>	ANN_SALARY
1		124		Mourgos		69600
2		141		Rajs		42000
3		142		Davies		37200
4		143		Matos		31200
5		144		Vargas		30000

# 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;
```

```
CREATE OR REPLACE VIEW succeeded.
```

- Column aliases in the CREATE OR REPLACE VIEW clause 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
  ON        (e.department_id = d.department_id)
  GROUP BY  d.department_name;
```

```
CREATE OR REPLACE VIEW succeeded.
```



# 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



# Rules for Performing DML Operations on a View

You cannot modify data in a view if it contains:

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

# Rules for Performing DML Operations on a View

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

- Group functions
- A GROUP BY clause
- The DISTINCT keyword
- The pseudocolumn ROWNUM keyword
- Columns defined by expressions
- NOT NULL columns in the base tables that are not selected by the view

# Using the WITH CHECK OPTION Clause

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

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

```
CREATE OR REPLACE VIEW succeeded.
```

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

# Denying DML Operations

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



# 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 ;
```

```
CREATE OR REPLACE VIEW succeeded.
```

# Removing a View

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

```
DROP VIEW view;
```

```
DROP VIEW empvu80;
```

```
DROP VIEW empvu80 succeeded.
```

# Practice 11: Overview of Part 1

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
- Removing views



# Lesson Agenda

- Overview of views:
  - Creating, modifying, and retrieving data from a view
  - DML operations on a view
  - Dropping a view
- Overview of sequences:
  - Creating, using, and modifying a sequence
  - Cache sequence values
  - NEXTVAL and CURRVAL pseudocolumns
- Overview of indexes
  - Creating, dropping indexes
- Overview of synonyms
  - Creating, dropping synonyms

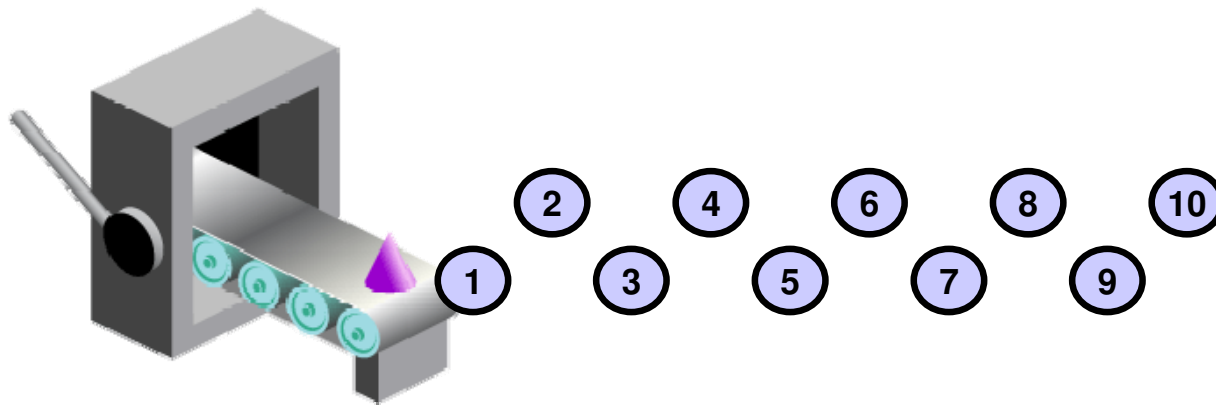
# Sequences

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 some queries
Synonym	Gives alternative names to objects

# 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



# CREATE SEQUENCE Statement: Syntax

Define a sequence to generate sequential numbers automatically:

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

# 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  
            INCREMENT BY 10  
            START WITH 120  
            MAXVALUE 9999  
            NOCACHE  
            NOCYCLE;
```

```
CREATE SEQUENCE succeeded.
```

# NEXTVAL and CURRVAL Pseudocolumns

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

# Using a Sequence

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

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

```
1 rows inserted
```

- View the current value for the DEPT\_DEPTID\_SEQ sequence:

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

# 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



# 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;
```

```
ALTER SEQUENCE dept_deptid_seq succeeded.
```

# 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;
```

```
DROP SEQUENCE dept_deptid_seq succeeded.
```

# Lesson Agenda

- Overview of views:
  - Creating, modifying, and retrieving data from a view
  - DML operations on a view
  - Dropping a view
- Overview of sequences:
  - Creating, using, and modifying a sequence
  - Cache sequence values
  - NEXTVAL and CURRVAL pseudocolumns
- Overview of indexes
  - Creating, dropping indexes
- Overview of synonyms
  - Creating, dropping synonyms

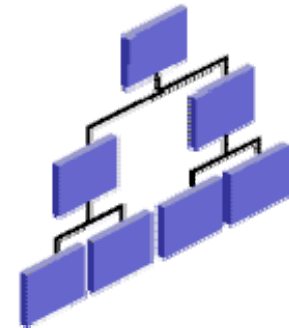
# Indexes

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 some queries
Synonym	Gives alternative names to objects

# Indexes

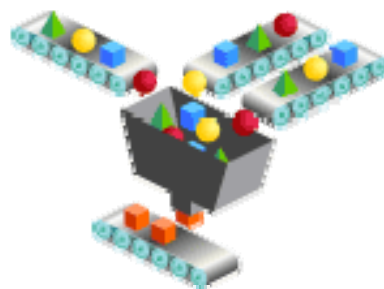
An index:

- Is a schema object
- May 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 independent of the table that it indexes
- Is used and maintained automatically by the Oracle server



# 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: Users can create nonunique indexes on columns to speed up access to the rows.



# Creating an Index

- Create an index on one or more columns:

```
CREATE [UNIQUE] [BITMAP] 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);
```

```
CREATE INDEX succeeded.
```

# Index Creation Guidelines

## Create an index when:

- |   |  |
|---|--|
| ✓ | A column contains a wide range of values   |
| ✓ | A column contains a large number of null values  |
| ✓ | One or more columns are frequently used together in a WHERE clause or a join condition                   |
| ✓ | The table is large and most queries are expected to retrieve less than 2% to 4% of the rows in the table |

## Do not create an index when:

- |   |   |
|---|---|
| ✗ | The columns are not often used as a condition in the query  |
| ✗ | The table is small or most queries are expected to retrieve more than 2% to 4% of the rows in the table |
| ✗ | The table is updated frequently   |
| ✗ | The indexed columns are referenced as part of an expression   |



# 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;
```

```
DROP INDEX emp_last_name_idx succeeded.
```

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

# Lesson Agenda

- Overview of views:
  - Creating, modifying, and retrieving data from a view
  - DML operations on a view
  - Dropping a view
- Overview of sequences:
  - Creating, using, and modifying a sequence
  - Cache sequence values
  - NEXTVAL and CURRVAL pseudocolumns
- Overview of indexes
  - Creating, dropping indexes
- Overview of synonyms
  - Creating, dropping synonyms

# Synonyms

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 some queries
Synonym	Gives alternative names to objects

# Creating a Synonym for an Object

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

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

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

# Creating and Removing Synonyms

- Create a shortened name for the DEPT\_SUM\_VU view:

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

```
CREATE SYNONYM succeeded.
```

- Drop a synonym:

```
DROP SYNONYM d_sum;
```

```
DROP SYNONYM d_sum succeeded.
```

# Quiz

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

1. True
2. False

# Summary

In this lesson, you should have learned how to:

- Create, use, and remove views
- Automatically generate sequence numbers by using a sequence generator
- Create indexes to improve speed of query retrieval
- Use synonyms to provide alternative names for objects

# Practice 11: Overview of Part 2

This practice covers the following topics:

- Creating sequences
- Using sequences
- Creating nonunique indexes
- Creating synonyms





# **Oracle Join Syntax**

# Objectives

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

- Write `SELECT` statements to access data from more than one table using equijoins and nonequijoins
- Join a table to itself by using a self-join
- View data that generally does not meet a join condition by using outer joins
- Generate a Cartesian product of all rows from two or more tables

# Obtaining Data from Multiple Tables

**EMPLOYEES**

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	100	King	90
2	101	Kochhar	90
3	102	De Haan	90
...			
18	202	Fay	20
19	205	Higgins	110
20	206	Gietz	110

**DEPARTMENTS**

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
1	10	Administration	1700
2	20	Marketing	1800
3	50	Shipping	1500
4	60	IT	1400
5	80	Sales	2500
6	90	Executive	1700
7	110	Accounting	1700
8	190	Contracting	1700

	EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
1	200	10	Administration
2	201	20	Marketing
3	202	20	Marketing
4	124	50	Shipping
5	144	50	Shipping
...			
18	205	110	Accounting
19	206	110	Accounting

# Cartesian Products

- A Cartesian product is formed when:
  - A join condition is omitted
  - A join condition is invalid
  - All rows in the first table are joined to all rows in the second table
- To avoid a Cartesian product, always include a valid join condition in a `WHERE` clause.

# Generating a Cartesian Product

**EMPLOYEES (20 rows)**

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	100	King	90
2	101	Kochhar	90
3	102	De Haan	90
4	103	Hunold	60
...			
19	205	Higgins	110
20	206	Gietz	110

**DEPARTMENTS (8 rows)**

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
1	10	Administration	1700
2	20	Marketing	1800
3	50	Shipping	1500
4	60	IT	1400
5	80	Sales	2500
6	90	Executive	1700
7	110	Accounting	1700
8	190	Contracting	1700

**Cartesian product:**  
**20 x 8 = 160 rows**

	EMPLOYEE_ID	DEPARTMENT_ID	LOCATION_ID
1	100	90	1700
2	101	90	1700
3	102	90	1700
4	103	60	1700
...			
159	205	110	1700
160	206	110	1700

# Types of Oracle-Proprietary Joins

- Equijoin
- Nonequijoin
- Outer join
- Self-join

# Joining Tables Using Oracle Syntax

Use a join to query data from more than one table:

```
SELECT    table1.column, table2.column
FROM      table1, table2
WHERE     table1.column1 = table2.column2;
```

- Write the join condition in the WHERE clause.
- Prefix the column name with the table name when the same column name appears in more than one table.

# Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Use table prefixes to improve performance.
- Instead of full table name prefixes, use table aliases.
- Table aliases give a table a shorter name.
  - Keeps SQL code smaller, uses less memory
- Use column aliases to distinguish columns that have identical names, but reside in different tables.



# Equijoins

**EMPLOYEES**

EMPLOYEE_ID	DEPARTMENT_ID
100	90
101	90
102	90
103	60
104	60
107	60
124	50
141	50
142	50
143	50
144	50
149	80
174	80
176	80

...

**DEPARTMENTS**

	DEPARTMENT_ID	DEPARTMENT_NAME
1	10	Administration
2	20	Marketing
3	50	Shipping
4	60	IT
5	80	Sales
6	90	Executive
7	110	Accounting
8	190	Contracting

**Foreign key**

**Primary key**

# Retrieving Records with Equijoins

```
SELECT e.employee_id, e.last_name, e.department_id,  
       d.department_id, d.location_id  
FROM   employees e, departments d  
WHERE  e.department_id = d.department_id;
```

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID_1	LOCATION_ID
1	200	Whalen	10	10	1700
2	201	Hartstein	20	20	1800
3	202	Fay	20	20	1800
4	124	Mourgos	50	50	1500
5	144	Vargas	50	50	1500
6	143	Matos	50	50	1500
7	142	Davies	50	50	1500
8	141	Rajs	50	50	1500
9	107	Lorentz	60	60	1400
10	104	Ernst	60	60	1400

...

# Retrieving Records with Equijoins: Example

```
SELECT d.department_id, d.department_name,  
       d.location_id, l.city  
FROM   departments d, locations l  
WHERE  d.location_id = l.location_id;
```

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID	CITY
1	60	IT	1400	Southlake
2	50	Shipping	1500	South San Francisco
3	10	Administration	1700	Seattle
4	90	Executive	1700	Seattle
5	110	Accounting	1700	Seattle
6	190	Contracting	1700	Seattle
7	20	Marketing	1800	Toronto
8	80	Sales	2500	Oxford

# Additional Search Conditions Using the AND Operator

```
SELECT  d.department_id, d.department_name, l.city
FROM    departments d, locations l
WHERE   d.location_id = l.location_id
AND d.department_id IN (20, 50);
```

	DEPARTMENT_ID	DEPARTMENT_NAME	CITY
1	20	Marketing	Toronto
2	50	Shipping	South San Francisco

# Joining More than Two Tables

**EMPLOYEES**

	LAST_NAME	DEPARTMENT_ID
1	King	90
2	Kochhar	90
3	De Haan	90
4	Hunold	60
5	Ernst	60
6	Lorentz	60
7	Mourgos	50
8	Rajs	50
9	Davies	50
10	Matos	50

**DEPARTMENTS**

DEPARTMENT_ID	LOCATION_ID
10	1700
20	1800
50	1500
60	1400
80	2500
90	1700
110	1700
190	1700

**LOCATIONS**

LOCATION_ID	CITY
1400	Southlake
1500	South San Francisco
1700	Seattle
1800	Toronto
2500	Oxford

...

To join  $n$  tables together, you need a minimum of  $n-1$  join conditions. For example, to join three tables, a minimum of two joins is required.

# Nonequijoins

**EMPLOYEES**

	LAST_NAME	SALARY
1	King	24000
2	Kochhar	17000
3	De Haan	17000
4	Hunold	9000
5	Ernst	6000
6	Lorentz	4200
7	Mourgos	5800
8	Rajs	3500
9	Davies	3100
10	Matos	2600
...		
19	Higgins	12000
20	Gietz	8300

**JOB\_GRADES**

	GRADE_LEVEL	LOWEST_SAL	HIGHEST_SAL
1	A	1000	2999
2	B	3000	5999
3	C	6000	9999
4	D	10000	14999
5	E	15000	24999
6	F	25000	40000

**JOB\_GRADES table defines LOWEST\_SAL and HIGHEST\_SAL range of values for each GRADE\_LEVEL. Hence, the GRADE\_LEVEL column can be used to assign grades to each employee.**

# Retrieving Records with Nonequijoins

```
SELECT e.last_name, e.salary, j.grade_level
FROM   employees e, job_grades j
WHERE  e.salary
      BETWEEN j.lowest_sal AND j.highest_sal;
```

	LAST_NAME	SALARY	GRADE_LEVEL
1	Vargas	2500	A
2	Matos	2600	A
3	Davies	3100	B
4	Rajs	3500	B
5	Lorentz	4200	B
6	Whalen	4400	B
7	Mourgos	5800	B
8	Ernst	6000	C
9	Fay	6000	C
10	Grant	7000	C

...

# Returning Records with No Direct Match with Outer Joins

**DEPARTMENTS**

DEPARTMENT_NAME	DEPARTMENT_ID
Administration	10
Marketing	20
Shipping	50
IT	60
Sales	80
Executive	90
Accounting	110
Contracting	190

**EMPLOYEES**

	DEPARTMENT_ID	LAST_NAME
1	90	King
2	90	Kochhar
3	90	De Haan
4	60	Hunold
5	60	Ernst
6	60	Lorentz
7	50	Mourgos
8	50	Rajs
9	50	Davies
10	50	Matos
...		
19	110	Higgins
20	110	Gietz

**There are no employees in department 190.**



# Outer Joins: Syntax

- You use an outer join to see rows that do not meet the join condition.
- The outer join operator is the plus sign (+).

```
SELECT table1.column, table2.column
FROM   table1, table2
WHERE  table1.column (+) = table2.column;
```

```
SELECT table1.column, table2.column
FROM   table1, table2
WHERE  table1.column = table2.column (+) ;
```

# Using Outer Joins

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e, departments d
WHERE  e.department_id(+) = d.department_id ;
```

	<small>AZ</small> LAST_NAME	<small>AZ</small> DEPARTMENT_ID	<small>AZ</small> DEPARTMENT_NAME
1	Whalen	10	Administration
2	Hartstein	20	Marketing
3	Fay	20	Marketing
4	Davies	50	Shipping
5	Vargas	50	Shipping
6	Rajs	50	Shipping
7	Mourgos	50	Shipping
8	Matos	50	Shipping
9	Hunold	60	IT
10	Ernst	60	IT

...

19	Gietz	110	Accounting
20	(null)	(null)	Contracting

# Outer Join: Another Example

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e, departments d
WHERE  e.department_id = d.department_id(+);
```

	A2	LAST_NAME	A2	DEPARTMENT_ID	A2	DEPARTMENT_NAME
1		Whalen		10		Administration
2		Fay		20		Marketing
3		Hartstein		20		Marketing
4		Vargas		50		Shipping
5		Matos		50		Shipping

...

17		King		90		Executive
18		Gietz		110		Accounting
19		Higgins		110		Accounting
20		Grant		(null)		(null)

# Joining a Table to Itself

**EMPLOYEES (WORKER)**

	EMPLOYEE_ID	LAST_NAME	MANAGER_ID
1	100	King	(null)
2	101	Kochhar	100
3	102	De Haan	100
4	103	Hunold	102
5	104	Ernst	103
6	107	Lorentz	103
7	124	Mourgos	100
8	141	Rajs	124
9	142	Davies	124
10	143	Matos	124

...

**EMPLOYEES (MANAGER)**

EMPLOYEE_ID	LAST_NAME
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst
107	Lorentz
124	Mourgos
141	Rajs
142	Davies
143	Matos

...

**MANAGER\_ID in the WORKER table is equal to  
EMPLOYEE\_ID in the MANAGER table.**

# Self-Join: Example

```
SELECT worker.last_name || ' works for '
       || manager.last_name
FROM   employees worker, employees manager
WHERE  worker.manager_id = manager.employee_id ;
```

	WORKER.LAST_NAME  'WORKSFOR'  MANAGER.LAST_NAME
1	Hunold works for De Haan
2	Fay works for Hartstein
3	Gietz works for Higgins
4	Lorentz works for Hunold
5	Ernst works for Hunold
6	Zlotkey works for King
7	Mourgos works for King
8	Kochhar works for King
9	Hartstein works for King
10	De Haan works for King

...

# Summary

In this appendix, you should have learned how to use joins to display data from multiple tables by using Oracle-proprietary syntax.

# Practice C: Overview

This practice covers the following topics:

- Joining tables by using an equijoin
- Performing outer and self-joins
- Adding conditions

# D

## Using SQL\*Plus

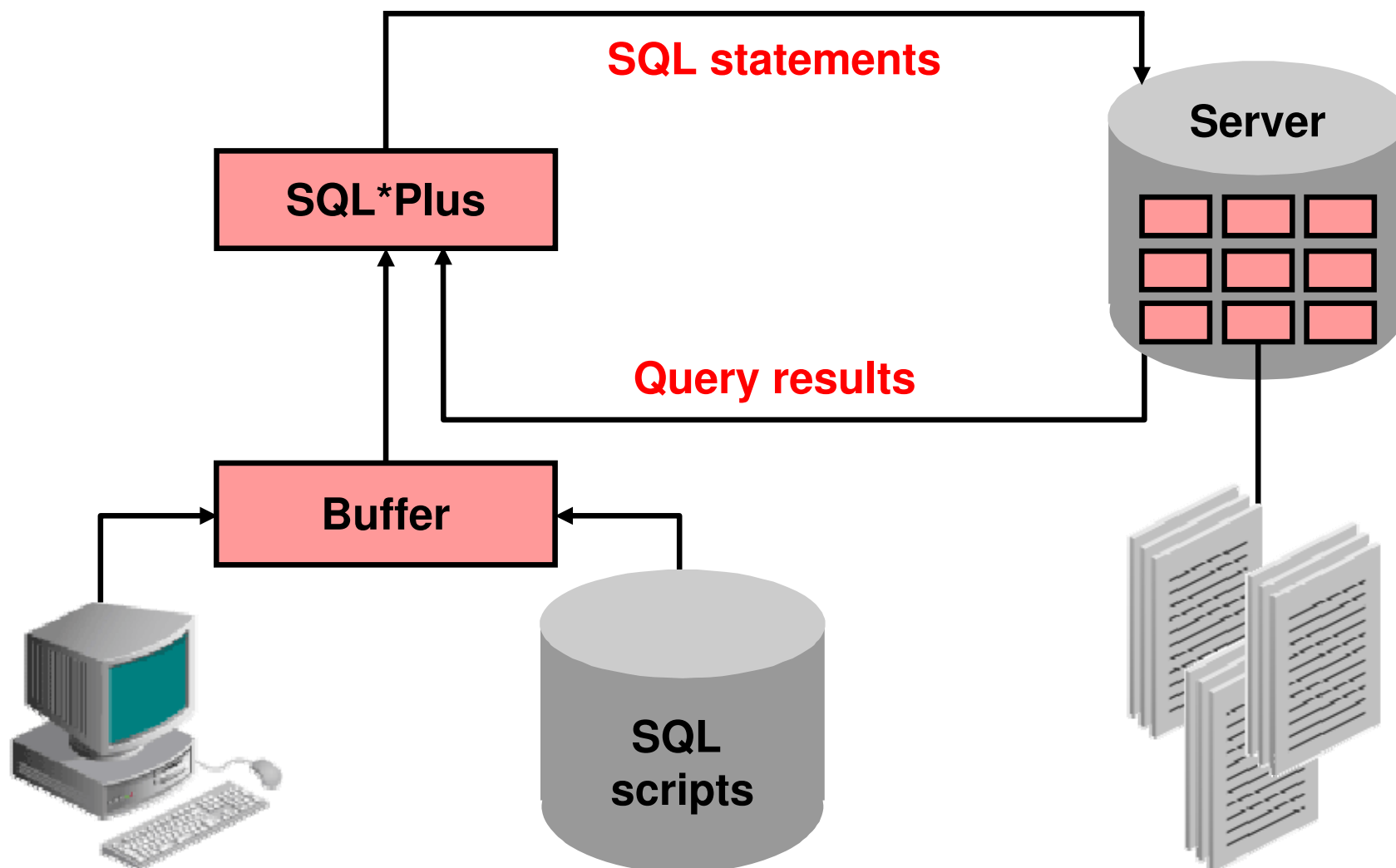


# Objectives

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

- Log in to SQL\*Plus
- Edit SQL commands
- Format output using SQL\*Plus commands
- Interact with script files

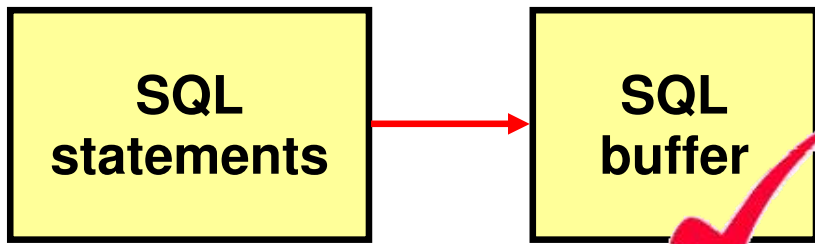
# SQL and SQL\*Plus Interaction



# SQL Statements Versus SQL\*Plus Commands

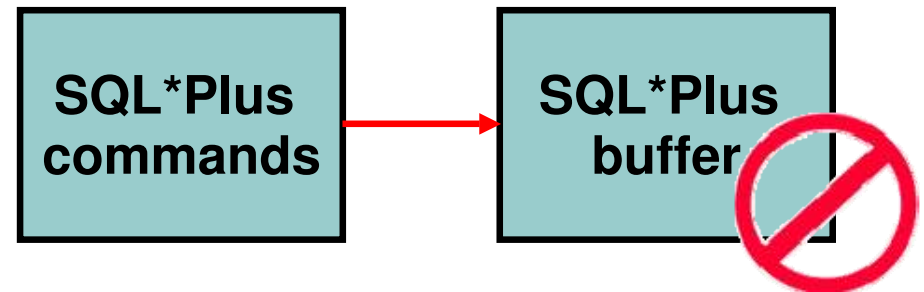
## SQL

- A language
- ANSI-standard
- Keywords cannot be abbreviated
- Statements manipulate data and table definitions in the database



## SQL\*Plus

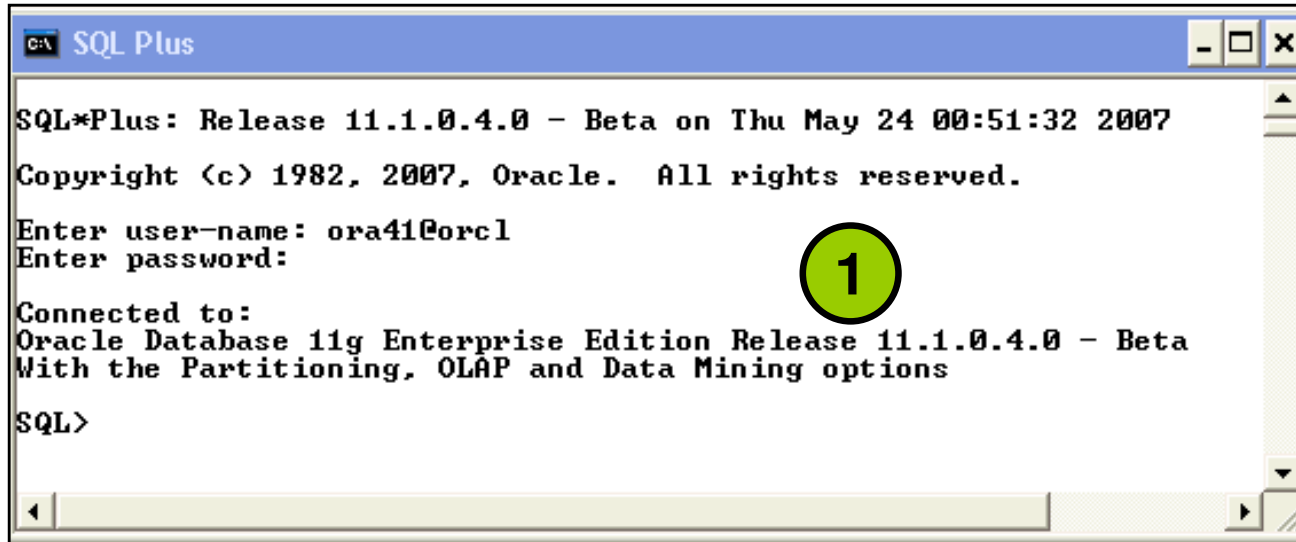
- An environment
- Oracle-proprietary
- Keywords can be abbreviated
- Commands do not allow manipulation of values in the database



# Overview of SQL\*Plus

- Log in to SQL\*Plus.
- Describe the table structure.
- Edit your SQL statement.
- Execute SQL from SQL\*Plus.
- Save SQL statements to files and append SQL statements to files.
- Execute saved files.
- Load commands from file to buffer to edit.

# Logging In to SQL\*Plus



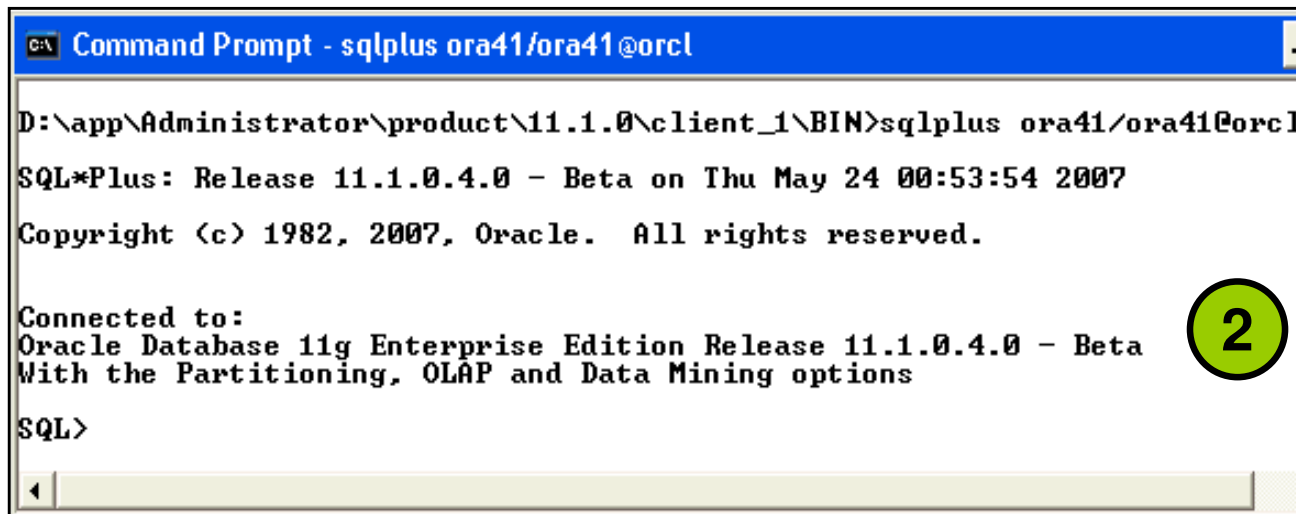
```
C:\ SQL Plus

SQL*Plus: Release 11.1.0.4.0 - Beta on Thu May 24 00:51:32 2007
Copyright (c) 1982, 2007, Oracle. All rights reserved.

Enter user-name: ora41@orcl
Enter password:
Connected to:
Oracle Database 11g Enterprise Edition Release 11.1.0.4.0 - Beta
With the Partitioning, OLAP and Data Mining options

SQL>
```

```
sqlplus [username[/password[@database]]]
```



```
C:\ Command Prompt - sqlplus ora41/ora41@orcl

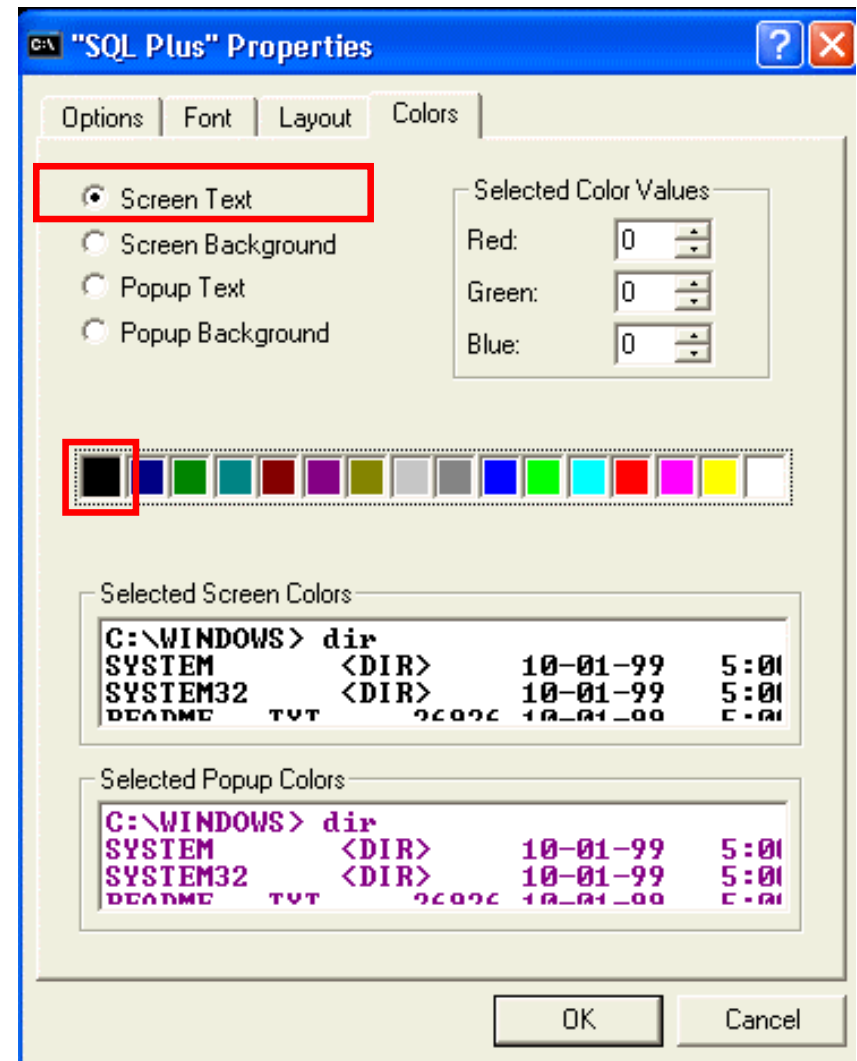
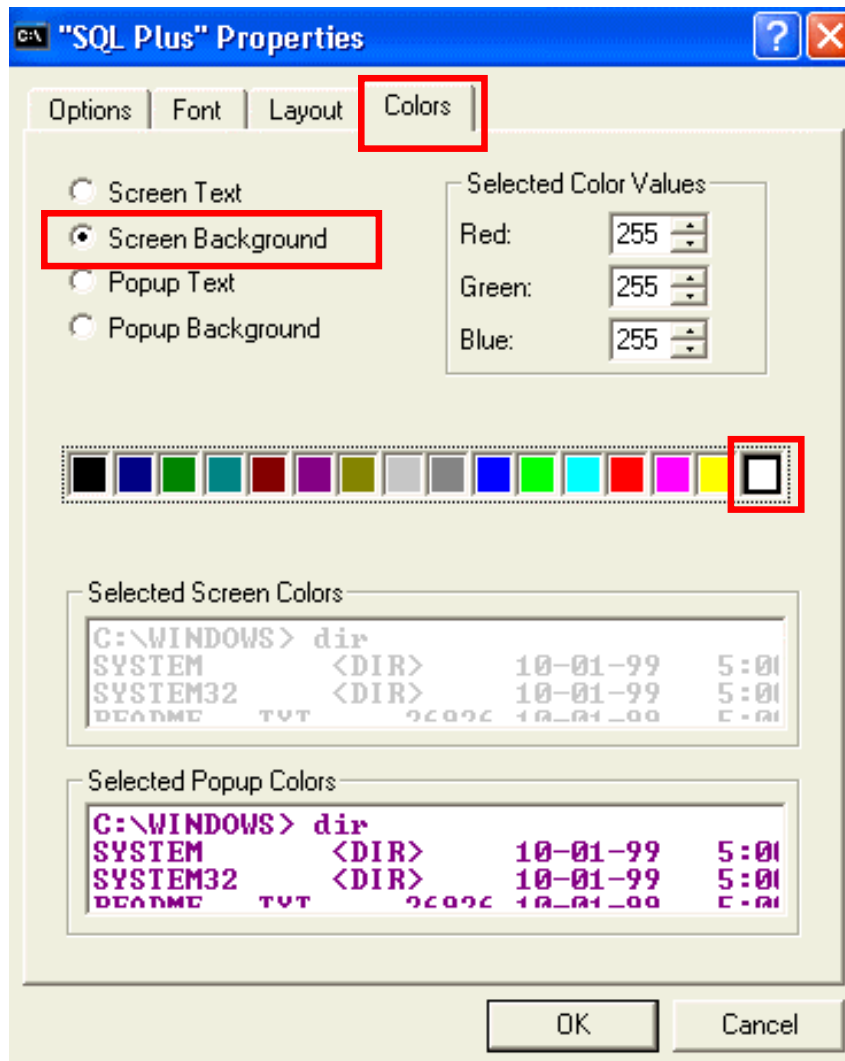
D:\app\Administrator\product\11.1.0\client_1\BIN>sqlplus ora41/ora41@orcl

SQL*Plus: Release 11.1.0.4.0 - Beta on Thu May 24 00:53:54 2007
Copyright (c) 1982, 2007, Oracle. All rights reserved.

Connected to:
Oracle Database 11g Enterprise Edition Release 11.1.0.4.0 - Beta
With the Partitioning, OLAP and Data Mining options

SQL>
```

# Changing the Settings of SQL\*Plus Environment



# Displaying Table Structure

Use the SQL\*Plus DESCRIBE command to display the structure of a table:

```
DESC[RIBE]  tablename
```

# Displaying Table Structure

```
DESCRIBE departments
```

Name	Null?	Type
-----	-----	-----
DEPARTMENT_ID	NOT NULL	NUMBER(4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(30)
MANAGER_ID		NUMBER(6)
LOCATION_ID		NUMBER(4)



# SQL\*Plus Editing Commands

- `A[PPEND] text`
- `C[HANGE] / old / new`
- `C[HANGE] / text /`
- `CL[EAR] BUFF[ER]`
- `DEL`
- `DEL n`
- `DEL m n`

# SQL\*Plus Editing Commands

- I [NPUT]
- I [NPUT] *text*
- L [IST]
- L [IST] *n*
- L [IST] *m n*
- R [UN]
- *n*
- *n text*
- *0 text*

# Using LIST, n, and APPEND

LIST

```
1  SELECT last_name  
2* FROM    employees
```

1

```
1* SELECT last_name
```

A , job\_id

```
1* SELECT last_name, job_id
```

LIST

```
1  SELECT last_name, job_id  
2* FROM    employees
```

# Using the CHANGE Command

```
LIST  
1* SELECT * from employees
```

```
c/employees/departments  
1* SELECT * from departments
```

```
LIST  
1* SELECT * from departments
```

# SQL\*Plus File Commands

- `SAVE filename`
- `GET filename`
- `START filename`
- `@ filename`
- `EDIT filename`
- `SPOOL filename`
- `EXIT`

# Using the SAVE, START, and EDIT Commands

```
LIST
```

```
1  SELECT last_name, manager_id, department_id
2*  FROM employees
```

```
SAVE my_query
```

```
Created file my_query
```

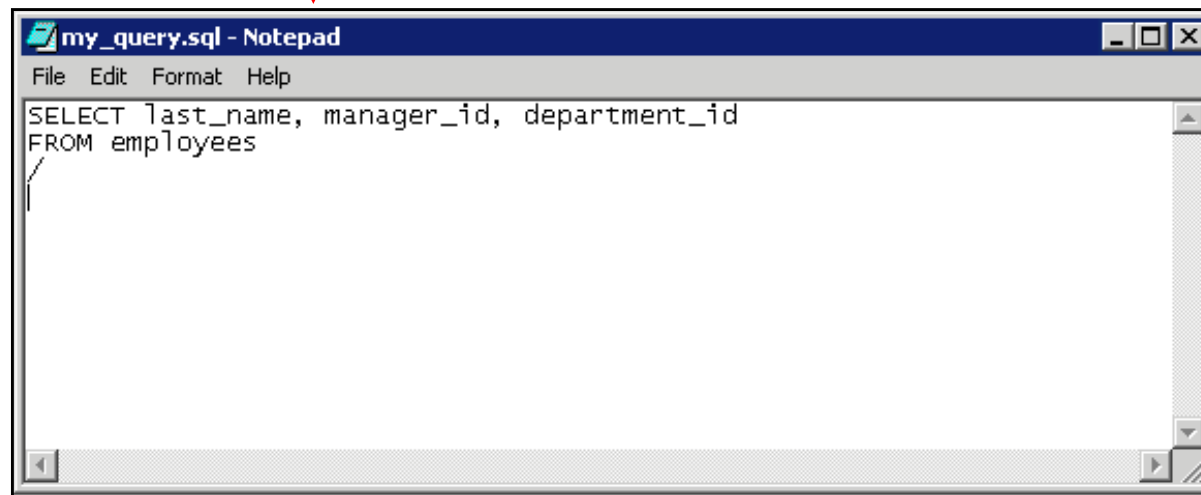
```
START my_query
```

LAST_NAME	MANAGER_ID	DEPARTMENT_ID
-----	-----	-----
King		90
Kochhar	100	90
...		

107 rows selected.

# Using the SAVE, START, and EDIT Commands

EDIT my\_query



# SERVEROUTPUT Command

- Use the `SET SERVEROUT [PUT]` command to control whether to display the output of stored procedures or PL/SQL blocks in SQL\*Plus.
- The `DBMS_OUTPUT` line length limit is increased from 255 bytes to 32767 bytes.
- The default size is now unlimited.
- Resources are not preallocated when `SERVEROUTPUT` is set.
- Because there is no performance penalty, use `UNLIMITED` unless you want to conserve physical memory.

```
SET SERVEROUT[PUT] {ON | OFF} [SIZE {n | UNL[IMITED]}]  
[FOR[MAT] {WRA[PPED] | WOR[D_WAPPED] | TRU[NCATED]}]
```



# Using the SQL\*Plus SPOOL Command

```
SPO[OL] [file_name[.ext] [CRE[ATE] | REP[LACE] |  
APP[END]] | OFF | OUT]
```

Option	Description
file_name[.ext]	Spools output to the specified file name
CRE[ATE]	Creates a new file with the name specified
REP[LACE]	Replaces the contents of an existing file. If the file does not exist, REPLACE creates the file.
APP[END]	Adds the contents of the buffer to the end of the file you specify
OFF	Stops spooling
OUT	Stops spooling and sends the file to your computer's standard (default) printer

# Using the AUTOTRACE Command

- Displays a report after the successful execution of SQL DML statements such as SELECT, INSERT, UPDATE, or DELETE
- May optionally include the query execution path and execution statistics

```
SET AUTOT[RACE] {ON | OFF | TRACE[ONLY]} [EXP[LAIN]]  
[STAT[ISTICS]]
```

```
SET AUTOTRACE ON
```

```
-- The AUTOTRACE report includes both the optimizer  
-- execution path and the SQL statement execution  
-- statistics
```

# Summary

In this appendix, you should have learned how to use SQL\*Plus as an environment to do the following:

- Execute SQL statements
- Edit SQL statements
- Format output
- Interact with script files



# **Using SQL Developer**

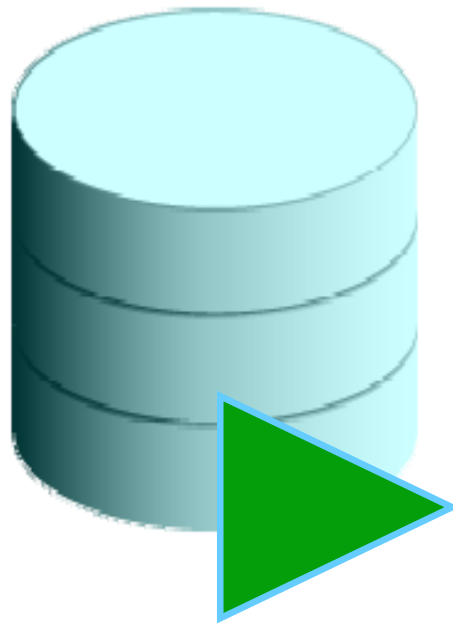
# Objectives

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

- List the key features of Oracle SQL Developer
- Install Oracle SQL Developer 1.2.1
- Identify menu items of Oracle SQL Developer
- Create a database connection
- Manage database objects
- Use SQL Worksheet
- Save and Run SQL scripts
- Create and save reports
- Install and use Oracle SQL Developer 1.5.3

# What Is Oracle SQL Developer?

- Oracle SQL Developer is a graphical tool that enhances productivity and simplifies database development tasks.
- You can connect to any target Oracle database schema by using standard Oracle database authentication.



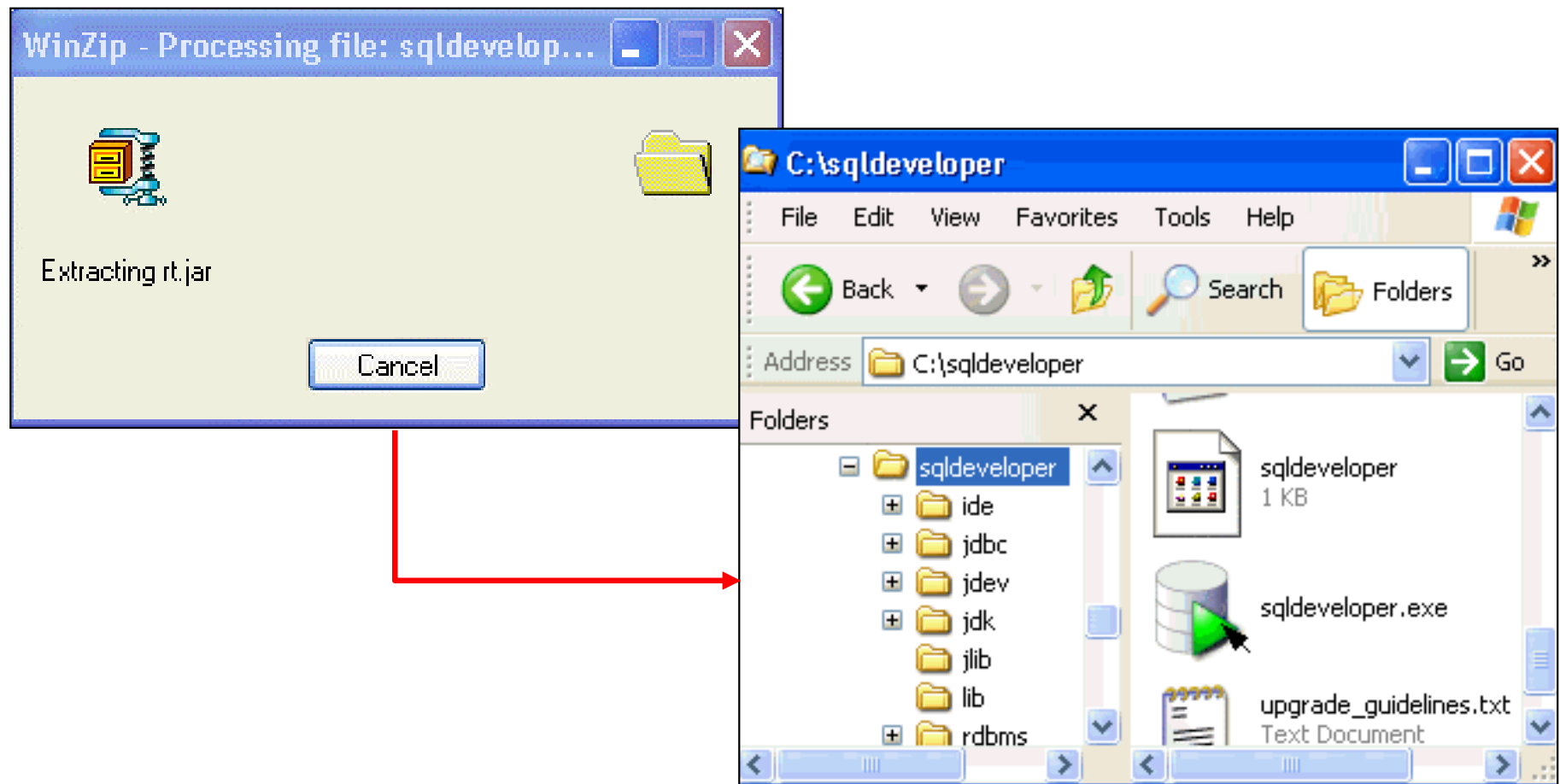
**SQL Developer**

# Specifications of SQL Developer

- Developed in Java
- Supports Windows, Linux, and Mac OS X platforms
- Default connectivity by using the JDBC Thin driver
- Does not require an installer
  - Unzip the downloaded SQL Developer kit and double-click `sqldeveloper.exe` to start SQL Developer.
- Connects to Oracle Database version 9.2.0.1 and later
- Freely downloadable from the following link:
  - [http://www.oracle.com/technology/products/database/sql\\_developer/index.html](http://www.oracle.com/technology/products/database/sql_developer/index.html)
- Needs JDK 1.5 installed on your system that can be downloaded from the following link:
  - [http://java.sun.com/javase/downloads/index\\_jdk5.jsp](http://java.sun.com/javase/downloads/index_jdk5.jsp)

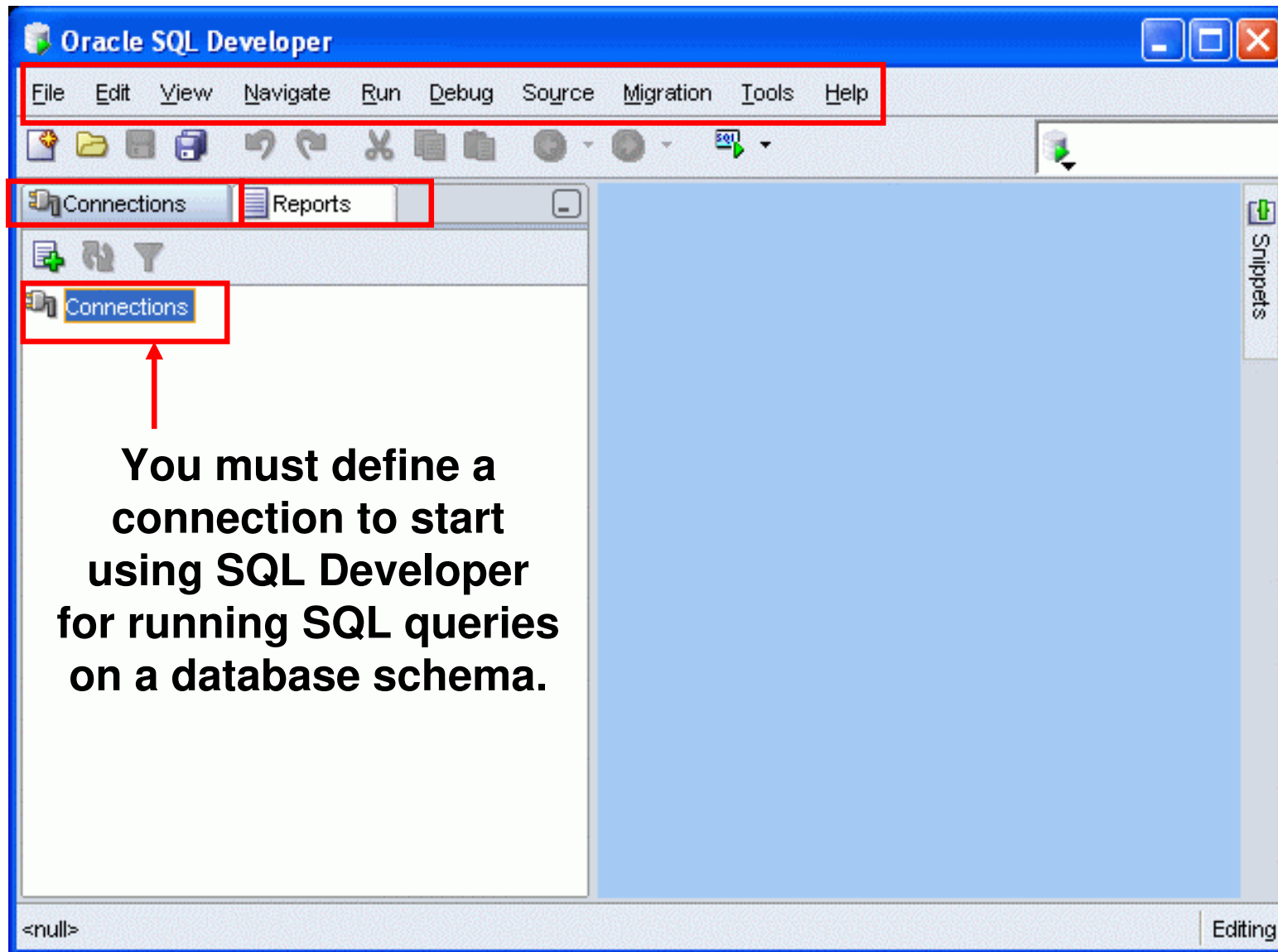
# Installing SQL Developer

Download the Oracle SQL Developer kit and unzip into any directory on your machine.





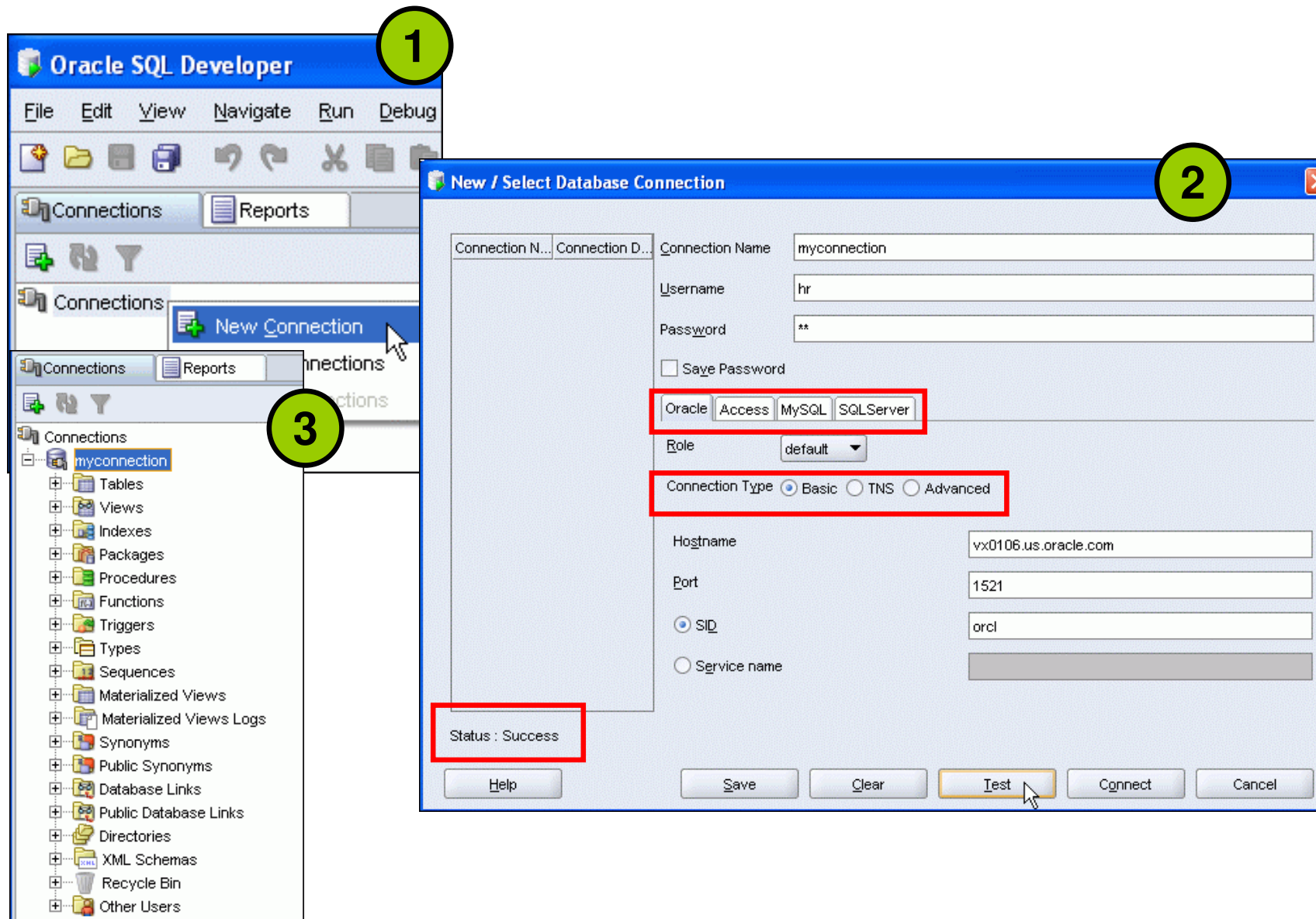
# SQL Developer 1.2 Interface



# Creating a Database Connection

- You must have at least one database connection to use SQL Developer.
- You can create and test connections for:
  - Multiple databases
  - Multiple schemas
- SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.
- You can export connections to an Extensible Markup Language (XML) file.
- Each additional database connection created is listed in the Connections Navigator hierarchy.

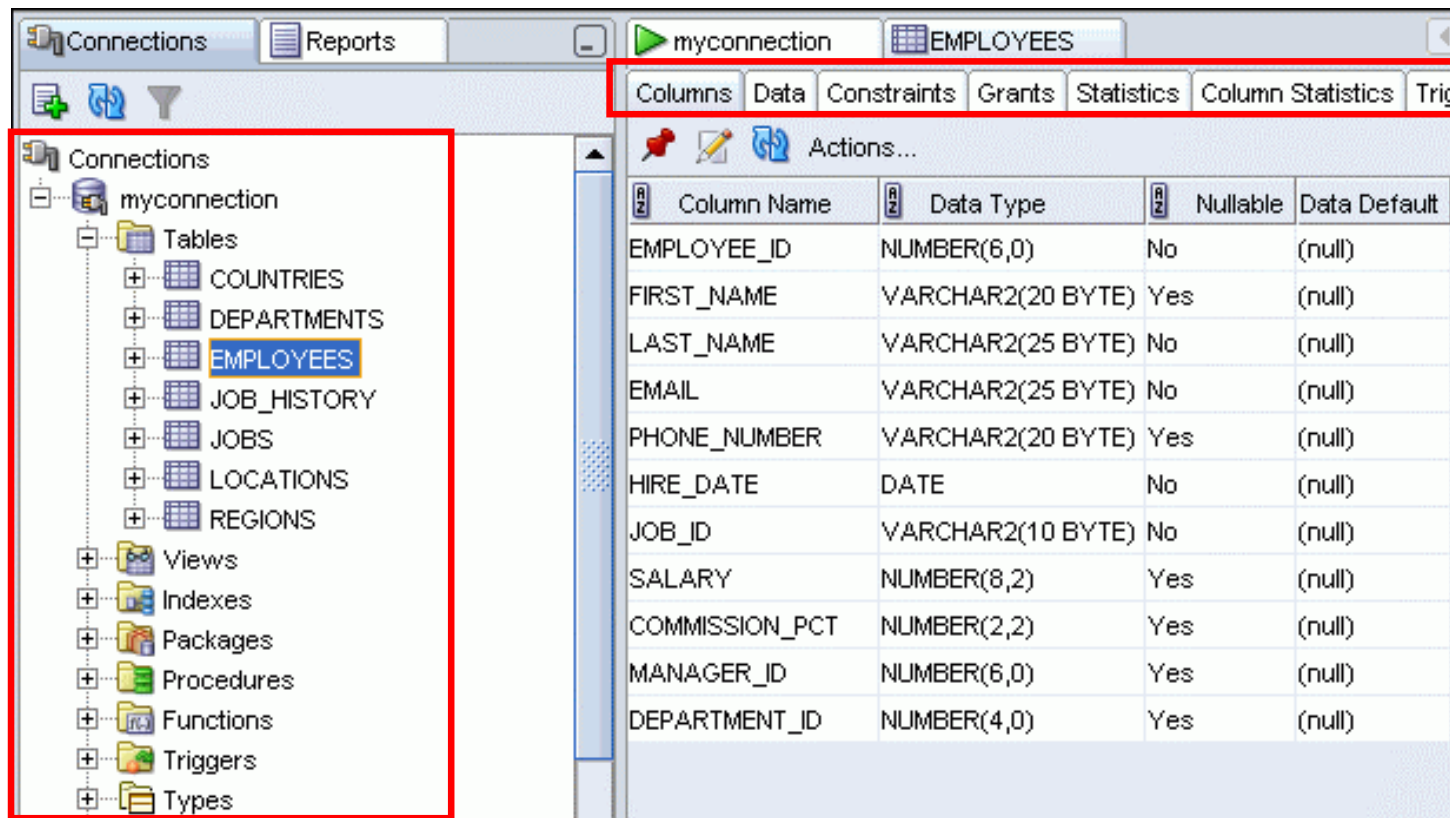
# Creating a Database Connection



# Browsing Database Objects

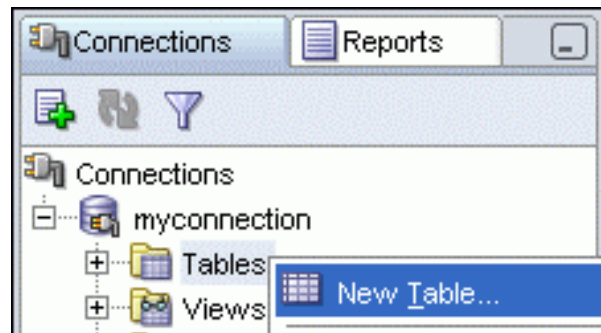
Use the Connections Navigator to:

- Browse through many objects in a database schema
- Review the definitions of objects at a glance



# Creating a Schema Object

- SQL Developer supports the creation of any schema object by:
  - Executing a SQL statement in SQL Worksheet
  - Using the context menu
- Edit the objects by using an edit dialog or one of the many context-sensitive menus.
- View the data definition language (DDL) for adjustments such as creating a new object or editing an existing schema object.





# Creating a New Table: Example

**Create Table**

Schema:

Name:

Table Type: ☒ Normal ☐ External ☐ Index Organized ☐ Temporary (Transaction) ☐ Temporary (Session)

☒ Advanced

**Columns**

- Primary Key
- Unique Constraints
- Foreign Keys
- Check Constraints
- Indexes
- Column Sequences
- Storage Options
- Lob Parameters
- Partitioning
  - Partition Definitions
  - Subpartition Templates
- Comment
- DDL

**Columns:**

- ID
- FIRST\_NAME
- LAST\_NAME
- RELATION
- BIRTHDATE

**Column Properties**

Name:

Datatype: ☒ Simple ☐ Complex

Type:

Precision:

Scale:

Default:

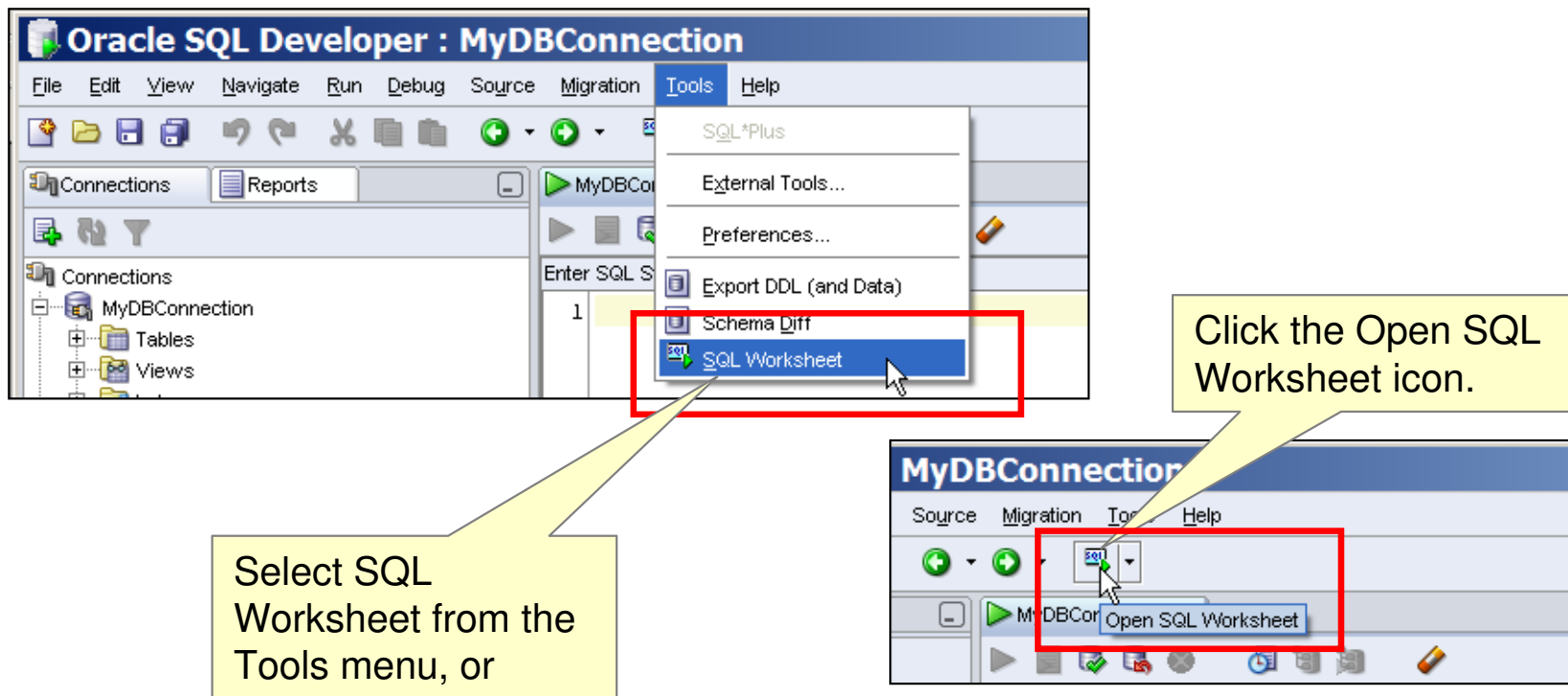
☒ Cannot be NULL

Comment:

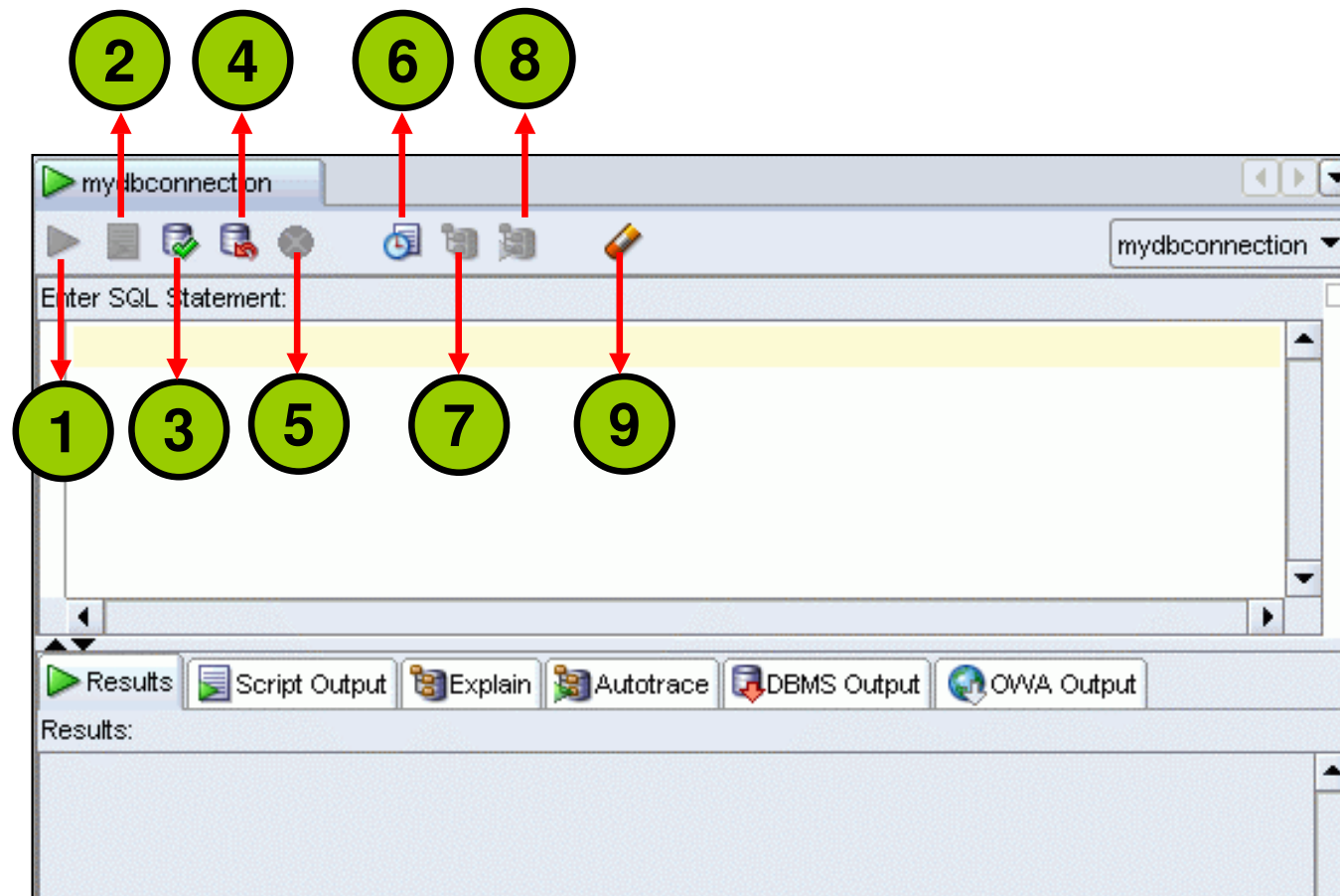
Help OK Cancel

# Using the SQL Worksheet

- Use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL \*Plus statements.
- Specify any actions that can be processed by the database connection associated with the worksheet.



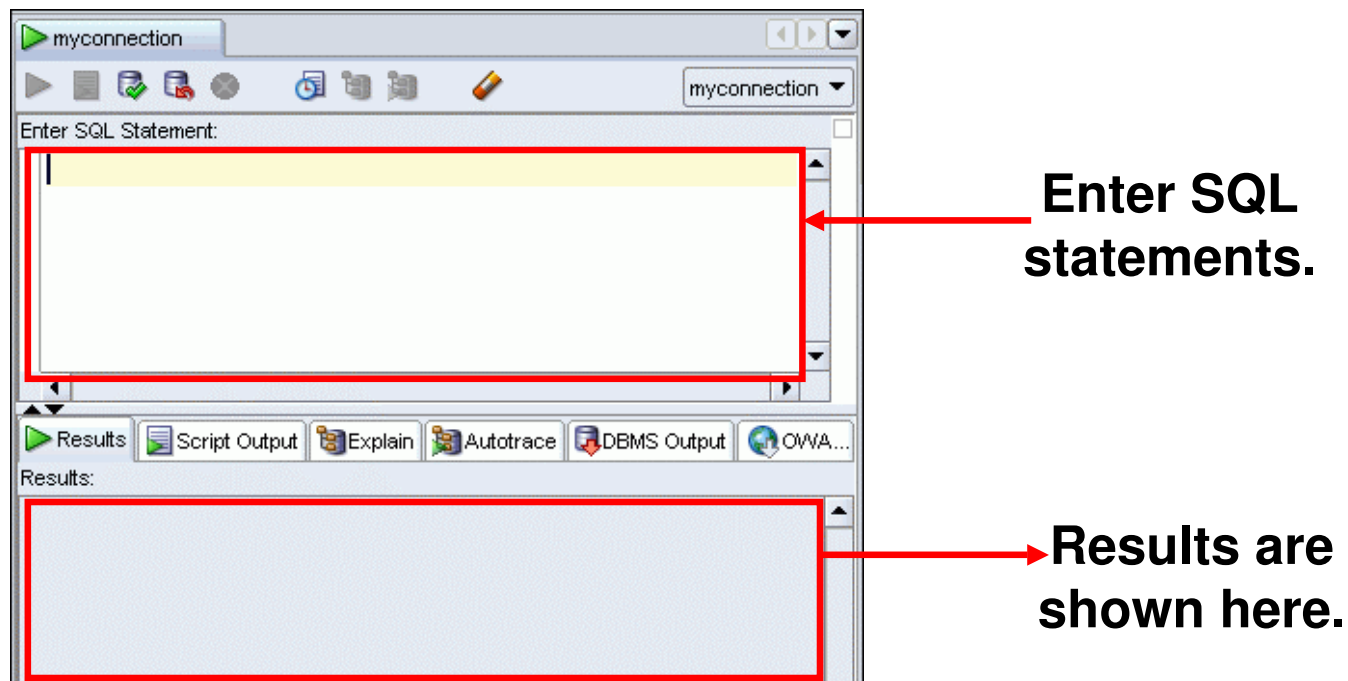
# Using the SQL Worksheet





# Using the SQL Worksheet

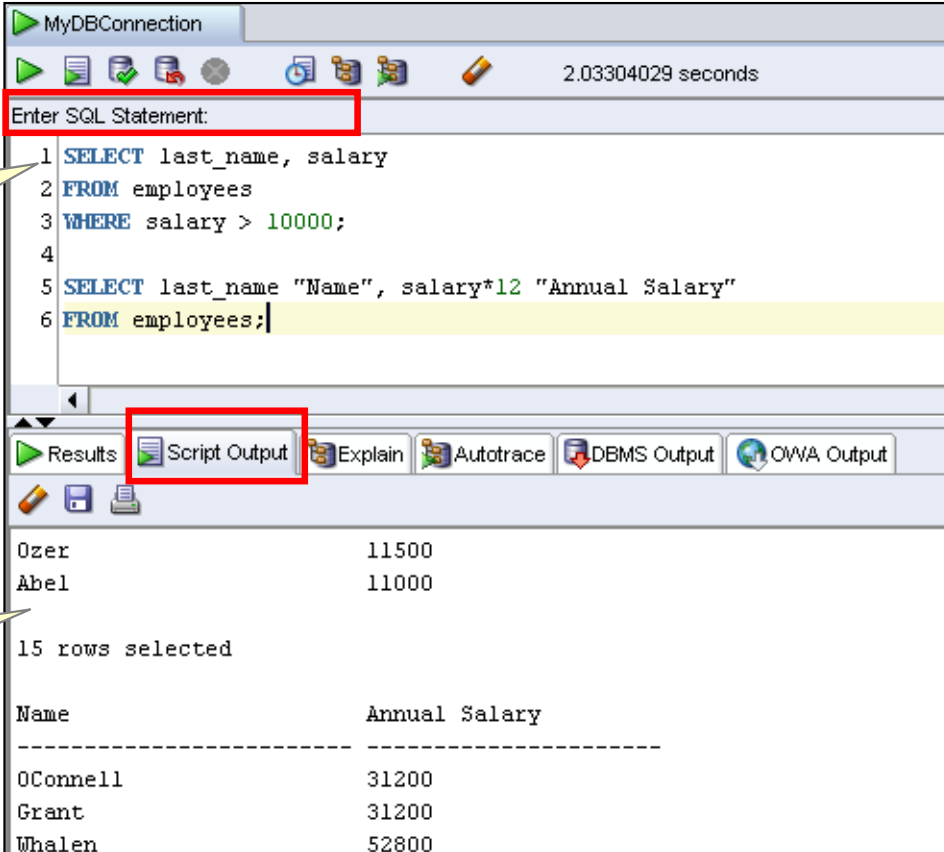
- Use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL\*Plus statements.
- Specify any actions that can be processed by the database connection associated with the worksheet.



# Executing SQL Statements

Use the Enter SQL Statement box to enter single or multiple SQL statements.

Use the Enter SQL Statement box to enter single or multiple SQL statements.



Script Output

Ozer	11500
Abel	11000

15 rows selected

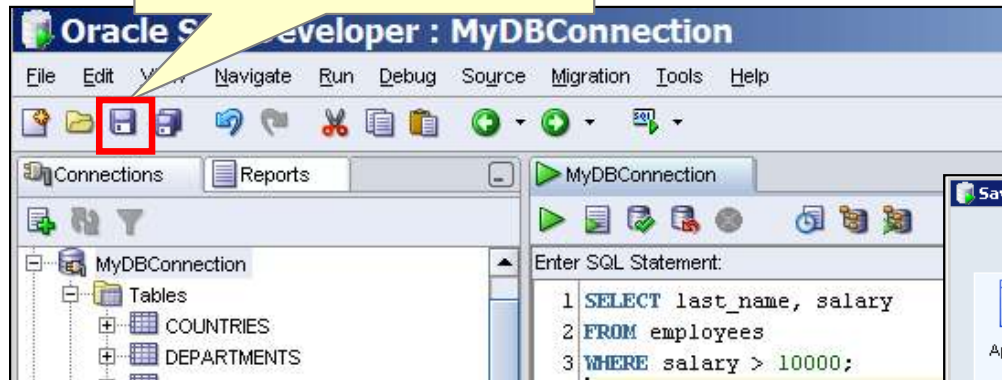
Name	Annual Salary
OConnell	31200
Grant	31200
Whalen	52800

View the results on the Script Output tabbed page.

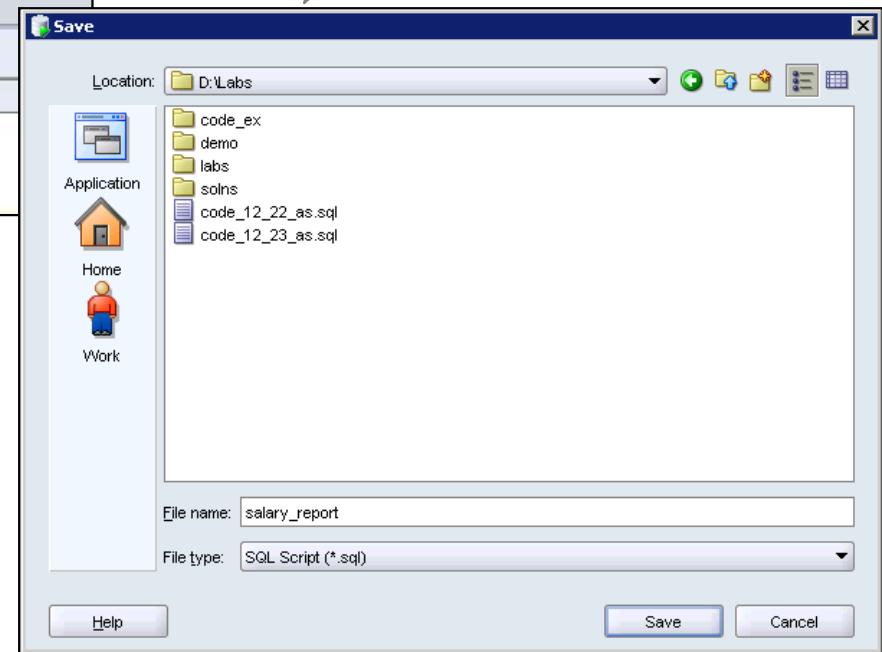
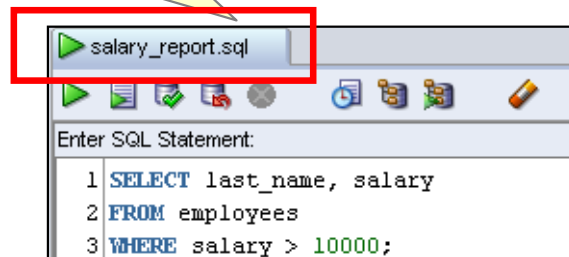
# Saving SQL Scripts

Click the Save icon to save your SQL statement to a file.

Enter a file name and identify a location to save the file, and click Save.



The contents of the saved file are visible and editable in your SQL Worksheet window.



# Executing Saved Script Files: Method 1

Right-click in the SQL Worksheet area, and select Open File from the shortcut menu.

Select (or navigate to) the script file that you want to open.

Click Open.

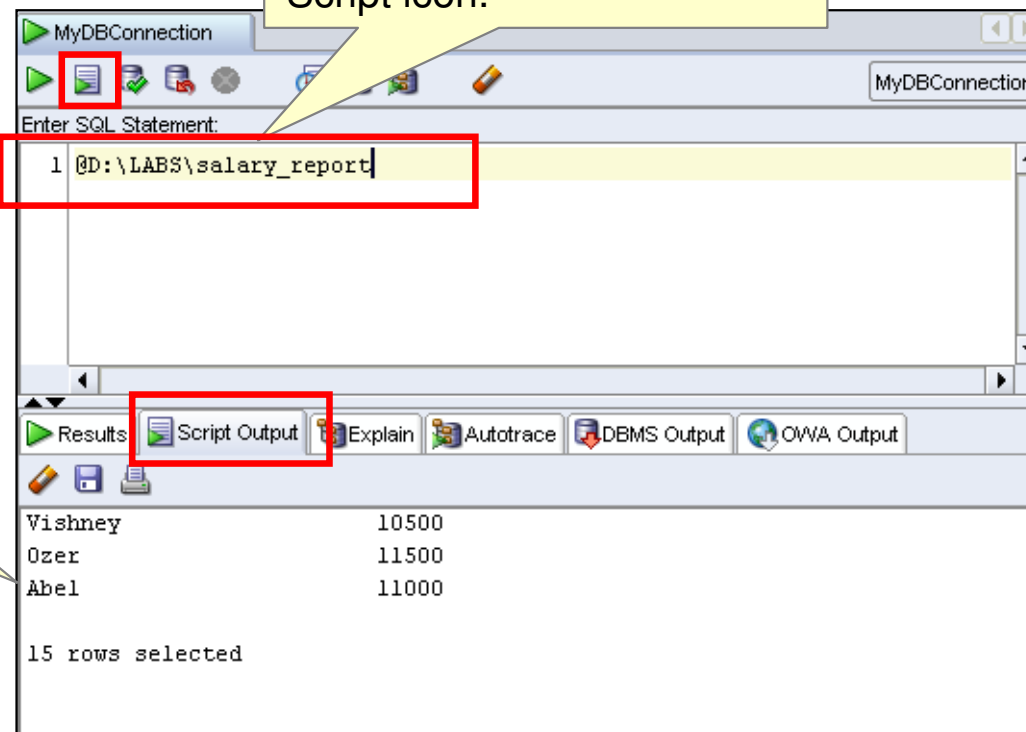
To run the code, click the Run Script (F5) icon.

```
CREATE PROCEDURE getemp IS -- header
emp_id employees.employee_id%type;
lname employees.last_name%type;
BEGIN
emp_id := 100;
SELECT last_name INTO lname
FROM EMPLOYEES
WHERE employee_id = emp_id;
DBMS_OUTPUT.PUT_LINE('Last name: '||lname);
END;
```

# Executing Saved Script Files: Method 2

Use the @ command followed by the location and name of the file you want to execute, and click the Run Script icon.

The output from the script is displayed on the Script Output tabbed page.



# Executing SQL Statements

Use the Enter SQL Statement box to enter single or multiple SQL statements.

The screenshot illustrates the workflow for executing an SQL statement in Oracle SQL Developer. It shows three main components: the 'Enter SQL Statement' box, the 'Results' window, and the 'Script Output' window.

**Enter SQL Statement Box:** This window contains the SQL query: `SELECT employee_id, last_name FROM employees;`. The 'Execute' button (a green play icon) is highlighted with a red box and labeled **F9**. The 'Script' button (a document icon) is also highlighted with a red box and labeled **F5**. A red arrow points from the 'Execute' button to the 'Results' window.

**Results Window:** This window displays the query results in a table format. The 'Results' button (a green play icon) is highlighted with a red box and labeled **F9**. The table shows the following data:

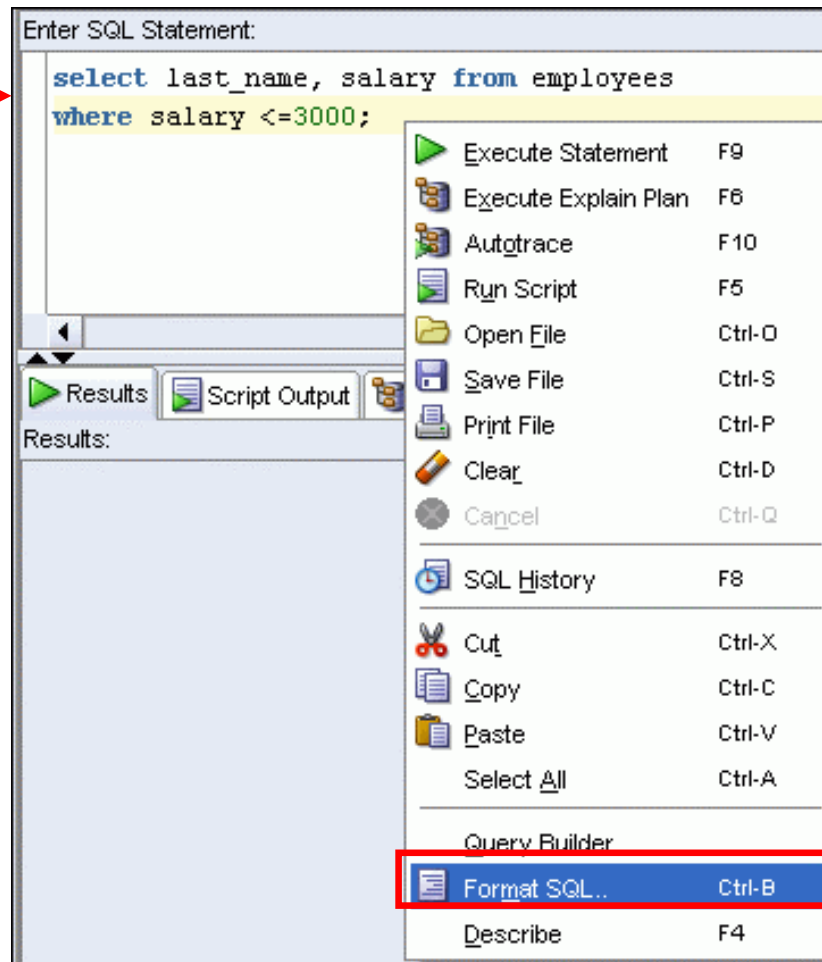
	EMPLOYEE_ID	LAST_NAME
1	100	King
2	101	Kochhar
3	102	De Haan
4	103	Hunold
5	104	Ernst

**Script Output Window:** This window displays the query results in a text format. The 'Script Output' button (a document icon) is highlighted with a red box and labeled **F5**. The text shows the following data:

EMPLOYEE_ID	LAST_NAME
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst
105	Austin

# Formatting the SQL Code

**Before  
formatting**

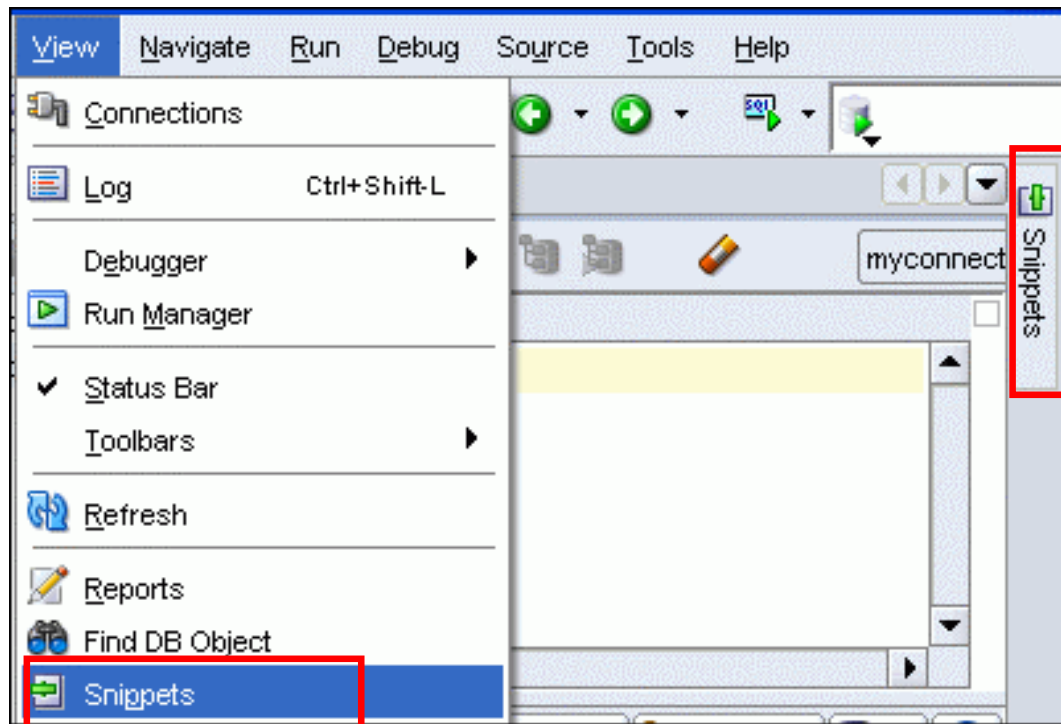


**After  
formatting**

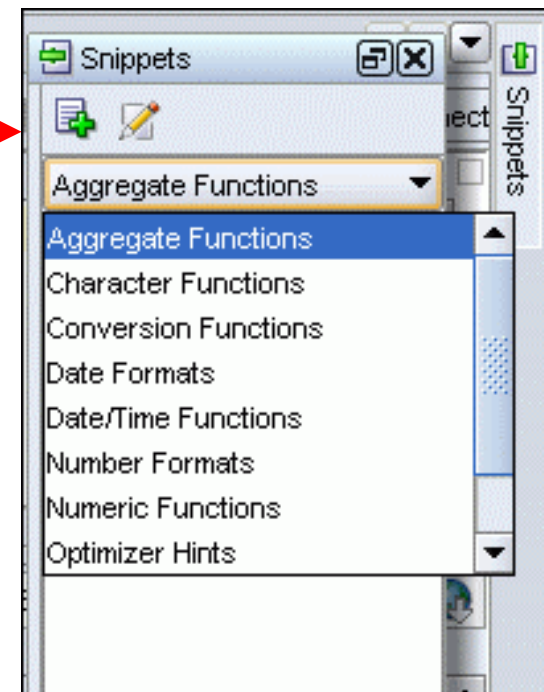
The screenshot shows the same SQL code after formatting: `SELECT last_name,  
salary  
FROM employees  
WHERE salary <= 3000;`. The code is now properly indented and uses uppercase keywords.

# Using Snippets

Snippets are code fragments that may be just syntax or examples.



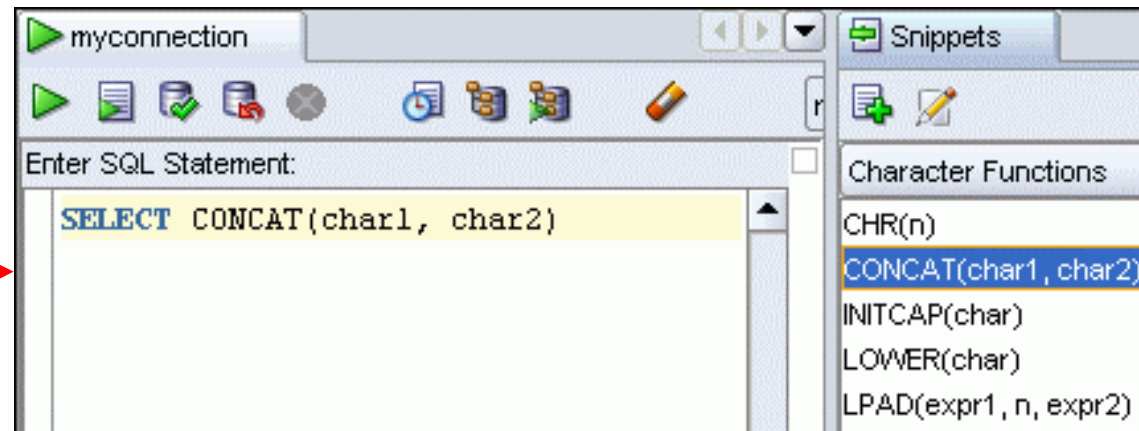
When you place your cursor here, it shows the Snippets window. From the drop-down list, you can select the functions category you want.



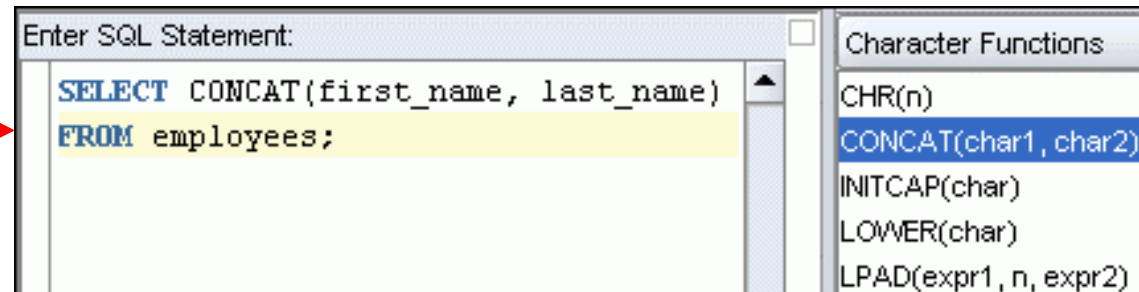


# Using Snippets: Example

**Inserting a snippet**

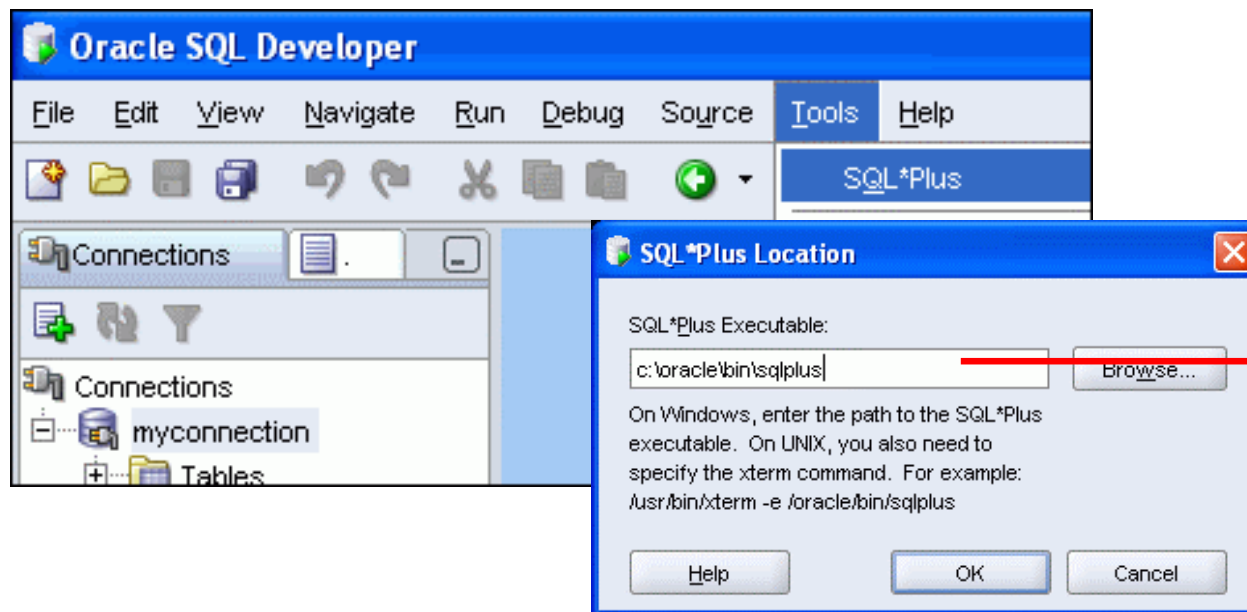


**Editing the snippet**



# Using SQL\*Plus

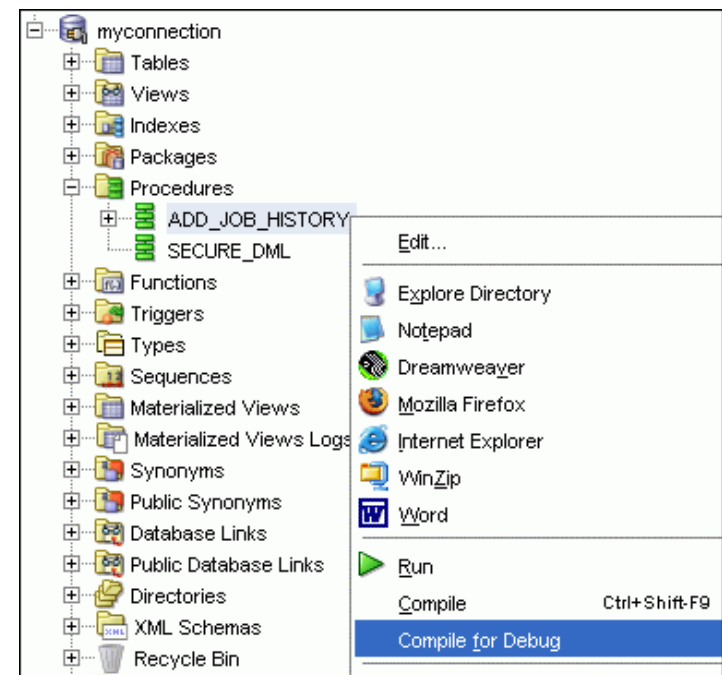
- You can invoke the SQL\*Plus command-line interface from SQL Developer.
- Close all the SQL Worksheets to enable the SQL\*Plus menu option.



**Provide the location of the sqlplus.exe file only the first time you invoke SQL\*Plus.**

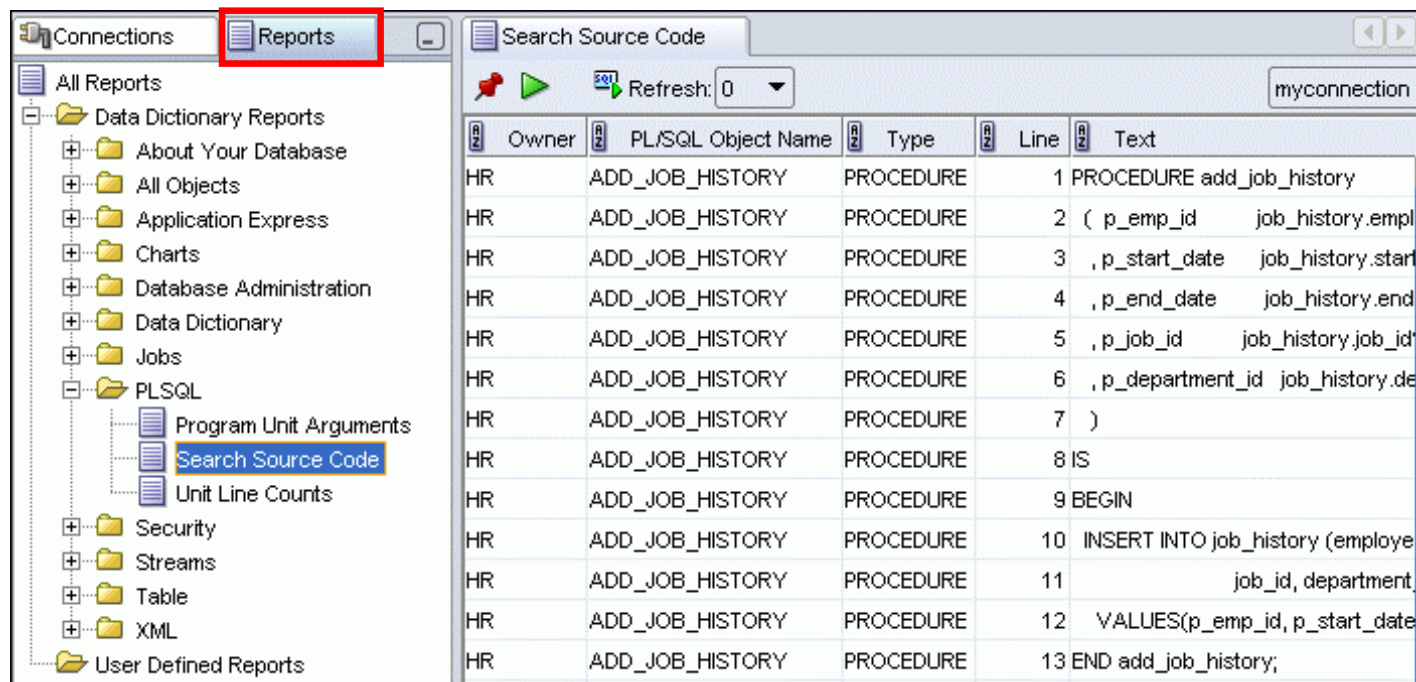
# Debugging Procedures and Functions

- Use SQL Developer to debug PL/SQL functions and procedures.
- Use the Compile for Debug option to perform a PL/SQL compilation so that the procedure can be debugged.
- Use Debug menu options to set breakpoints, and to perform step into, step over tasks.



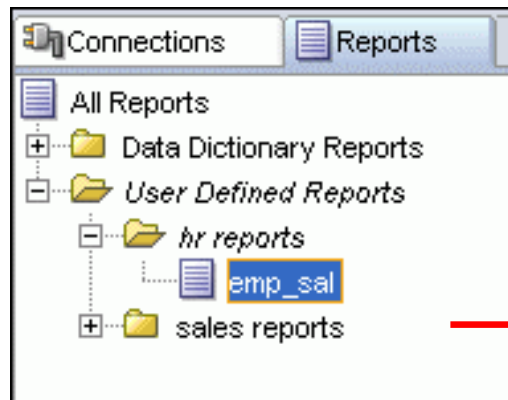
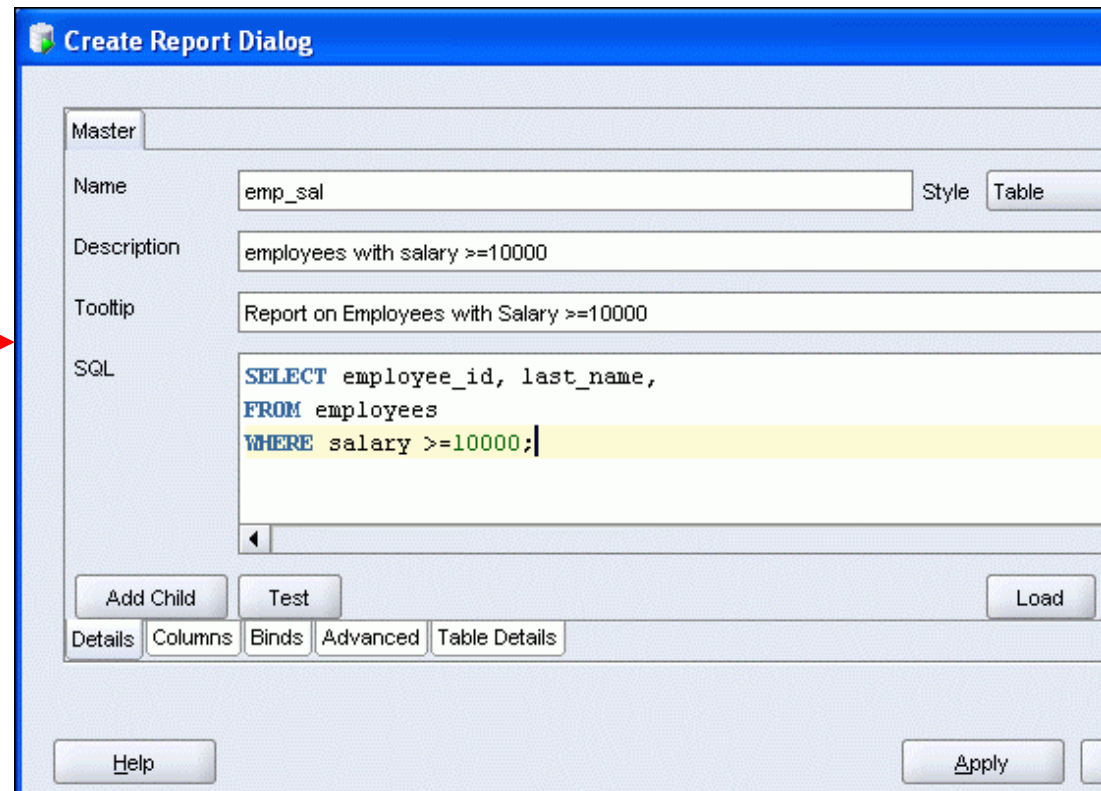
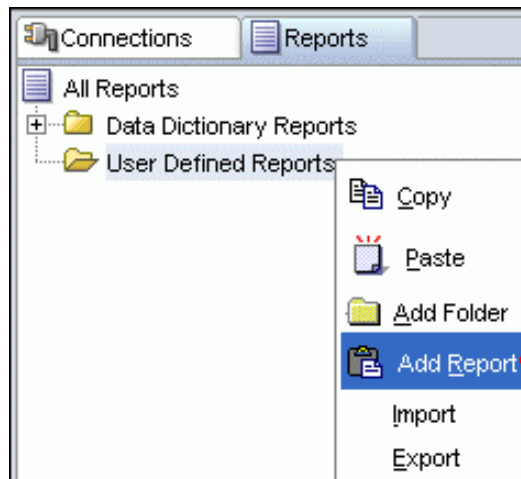
# Database Reporting

SQL Developer provides a number of predefined reports about the database and its objects.



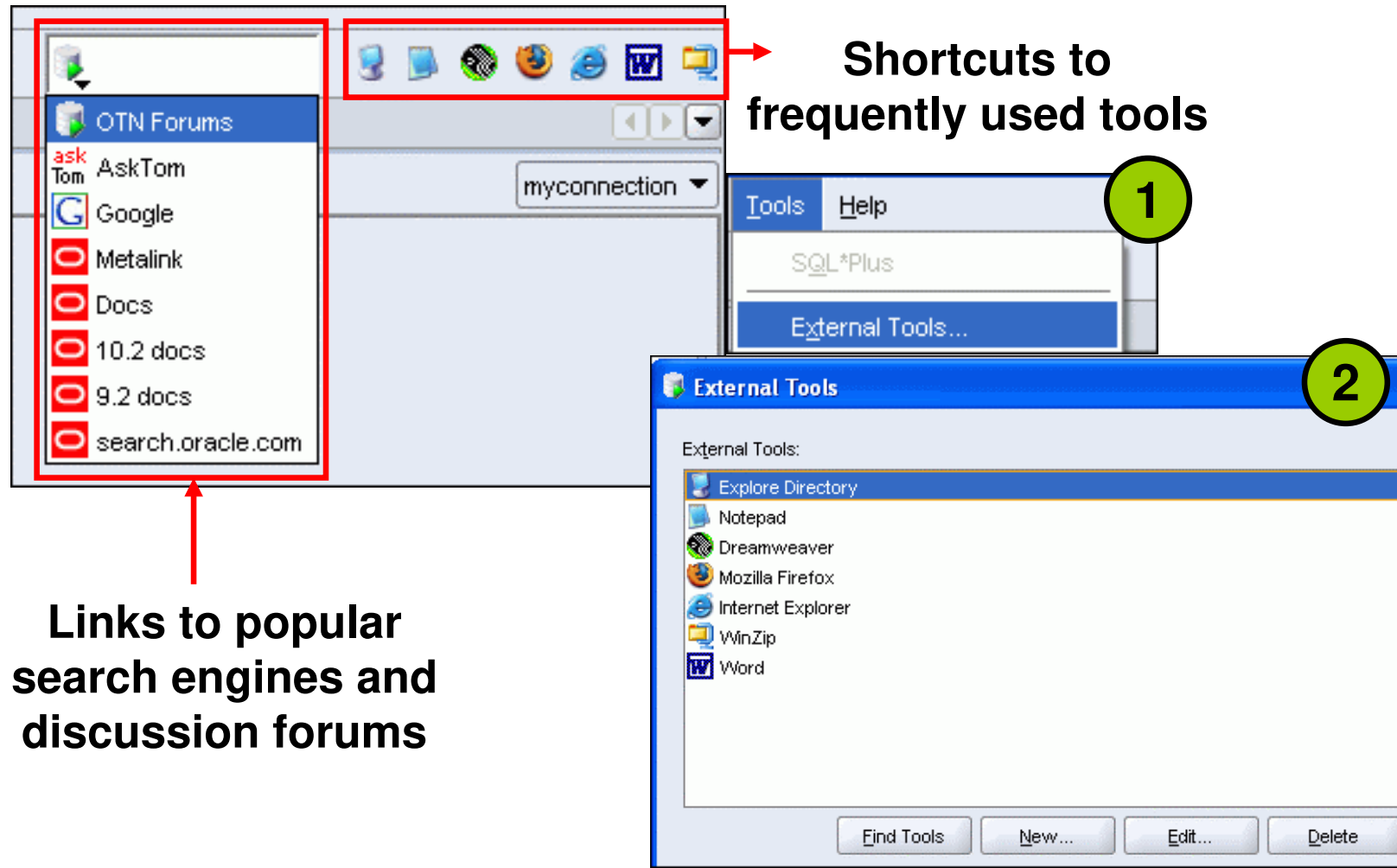
# Creating a User-Defined Report

Create and save user-defined reports for repeated use.



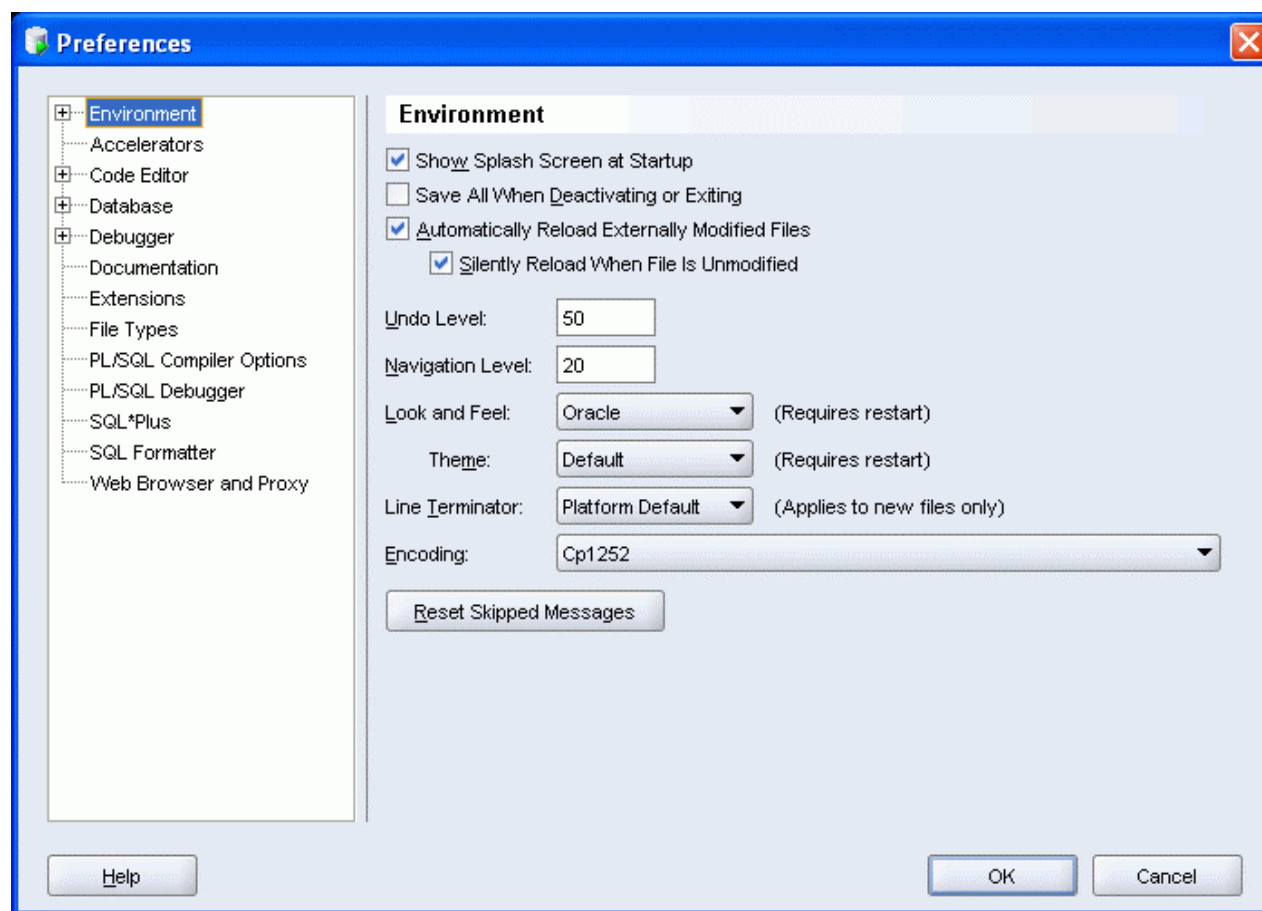
**Organize reports in folders.**

# Search Engines and External Tools



# Setting Preferences

- Customize the SQL Developer interface and environment.
- In the Tools menu, select Preferences.





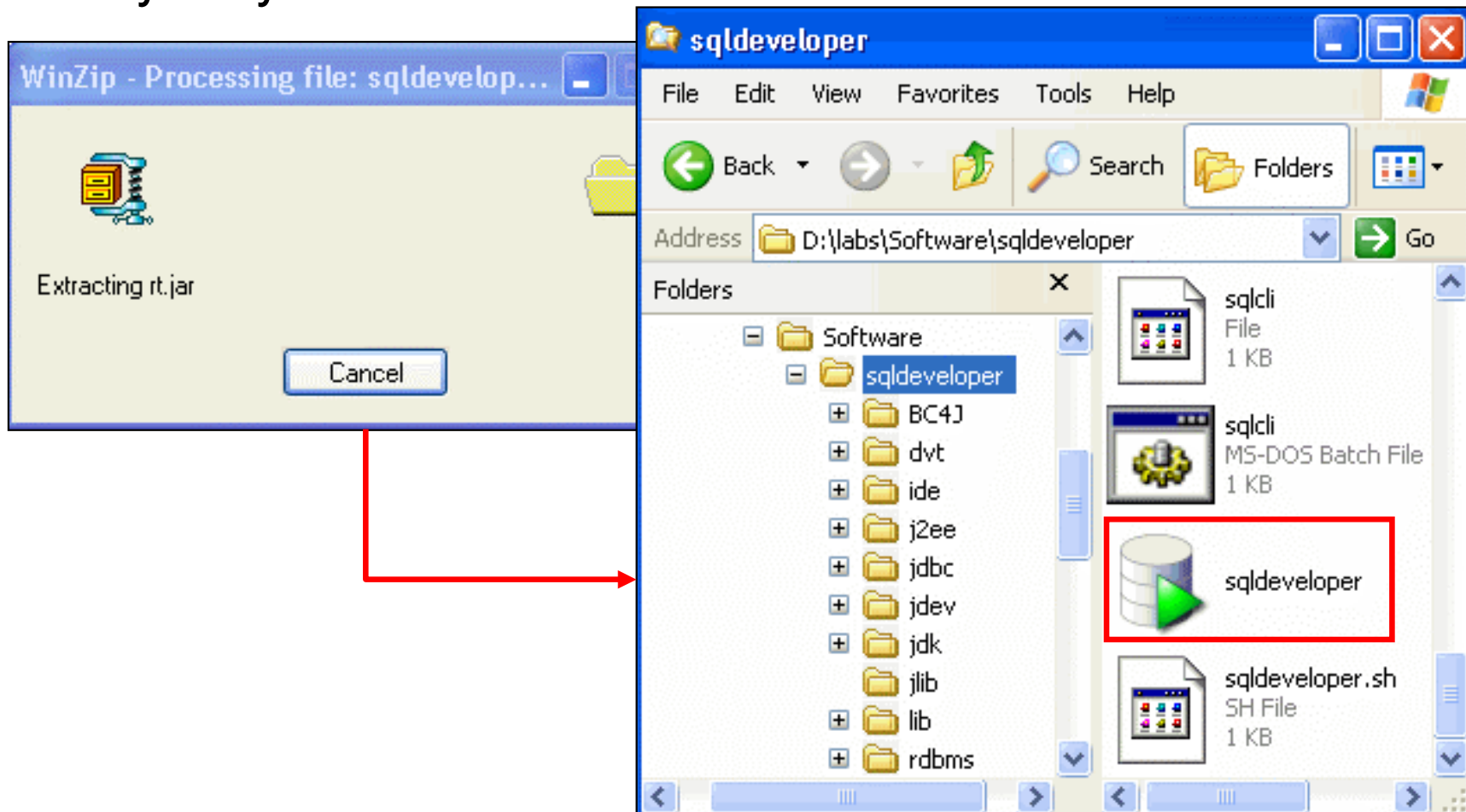
# Specifications of SQL Developer 1.5.3

- SQL Developer 1.5.3 is the first translation release, and is a patch to Oracle SQL Developer 1.5.
- New feature list is available at:
  - [http://www.oracle.com/technology/products/database/sql\\_developer/files/newFeatures\\_v15.html](http://www.oracle.com/technology/products/database/sql_developer/files/newFeatures_v15.html)
- Supports Windows, Linux, and Mac OS X platforms
- To install, unzip the downloaded SQL Developer kit, which includes the required minimum JDK (JDK1.5.0\_06).
- To start, double-click `sqldeveloper.exe`
- Connects to Oracle Database version 9.2.0.1 and later
- Freely downloadable from the following link:
  - [http://www.oracle.com/technology/products/database/sql\\_developer/index.html](http://www.oracle.com/technology/products/database/sql_developer/index.html)

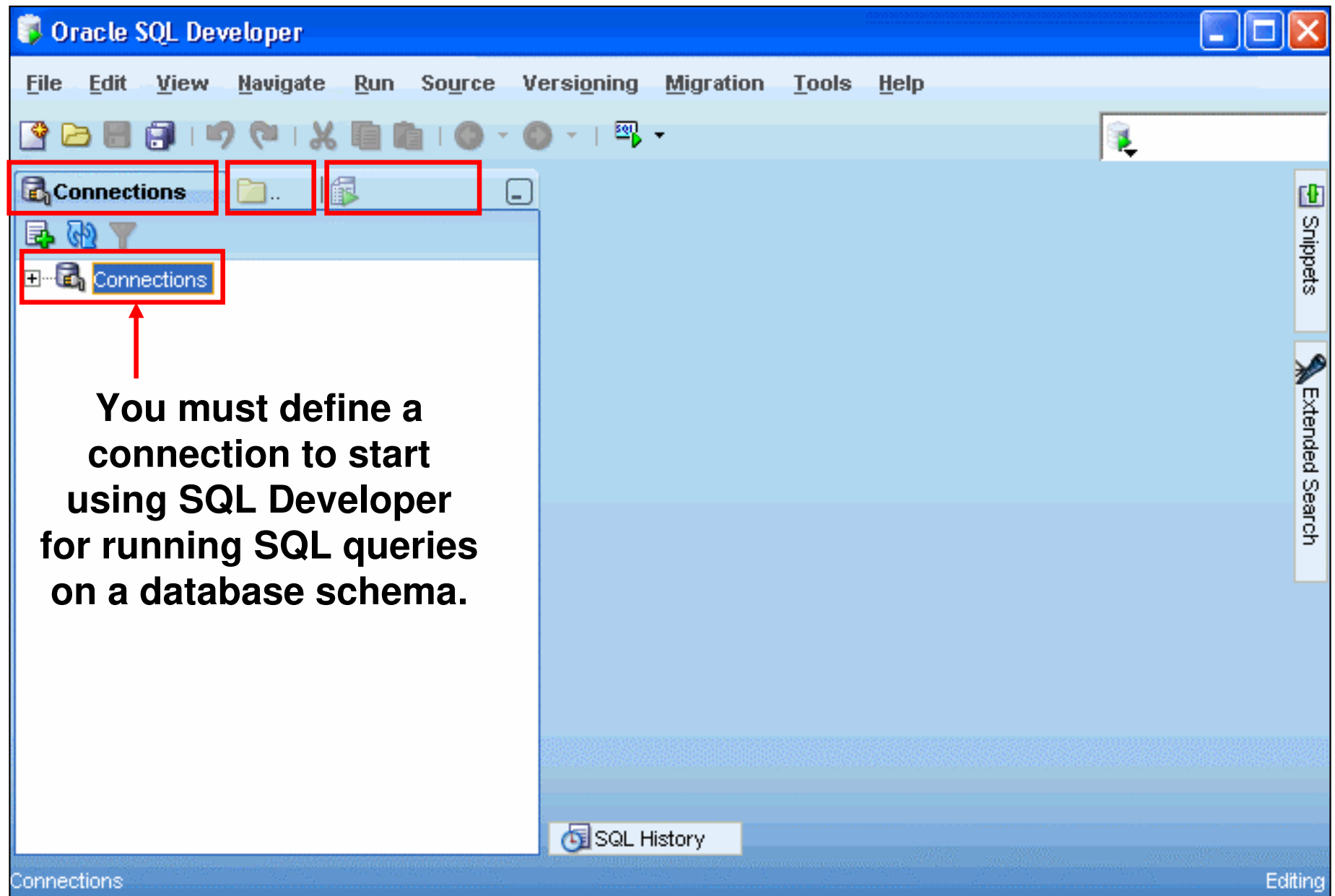


# Installing SQL Developer 1.5.3

Download the Oracle SQL Developer kit and unzip into any directory on your machine.



# SQL Developer 1.5.3 Interface



# Summary

In this appendix, you should have learned how to use SQL Developer to do the following:

- Browse, create, and edit database objects
- Execute SQL statements and scripts in SQL Worksheet
- Create and save custom reports