Oracle Database 10*g*: SQL Fundamentals I

Electronic Presentation

D17108GC11
Production 1.1
August 2004
D39769

ORACLE®

Author

Nancy Greenberg

Technical Contributors and Reviewers

Wayne Abbott
Christian Bauwens
Perry Benson
Brian Boxx
Zarko Cesljas
Dairy Chan
Laszlo Czinkoczki
Marjolein Dekkers
Matthew Gregory
Stefan Grenstad
Joel Goodman
Rosita Hanoman
Sushma Jagannath
Angelika Krupp
Christopher Lawless
Marcelo Manzano
Isabelle Marchand
Malika Marghadi
Valli Pataballa
Elspeth Payne
Ligia Jasmin Robayo
Bryan Roberts
Helen Robertson
Lata Shivaprasad
John Soltani
Priya Vennapusa
Ken Woolfe

Publisher

Jobi Varghese

Introduction

Lesson Objectives

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

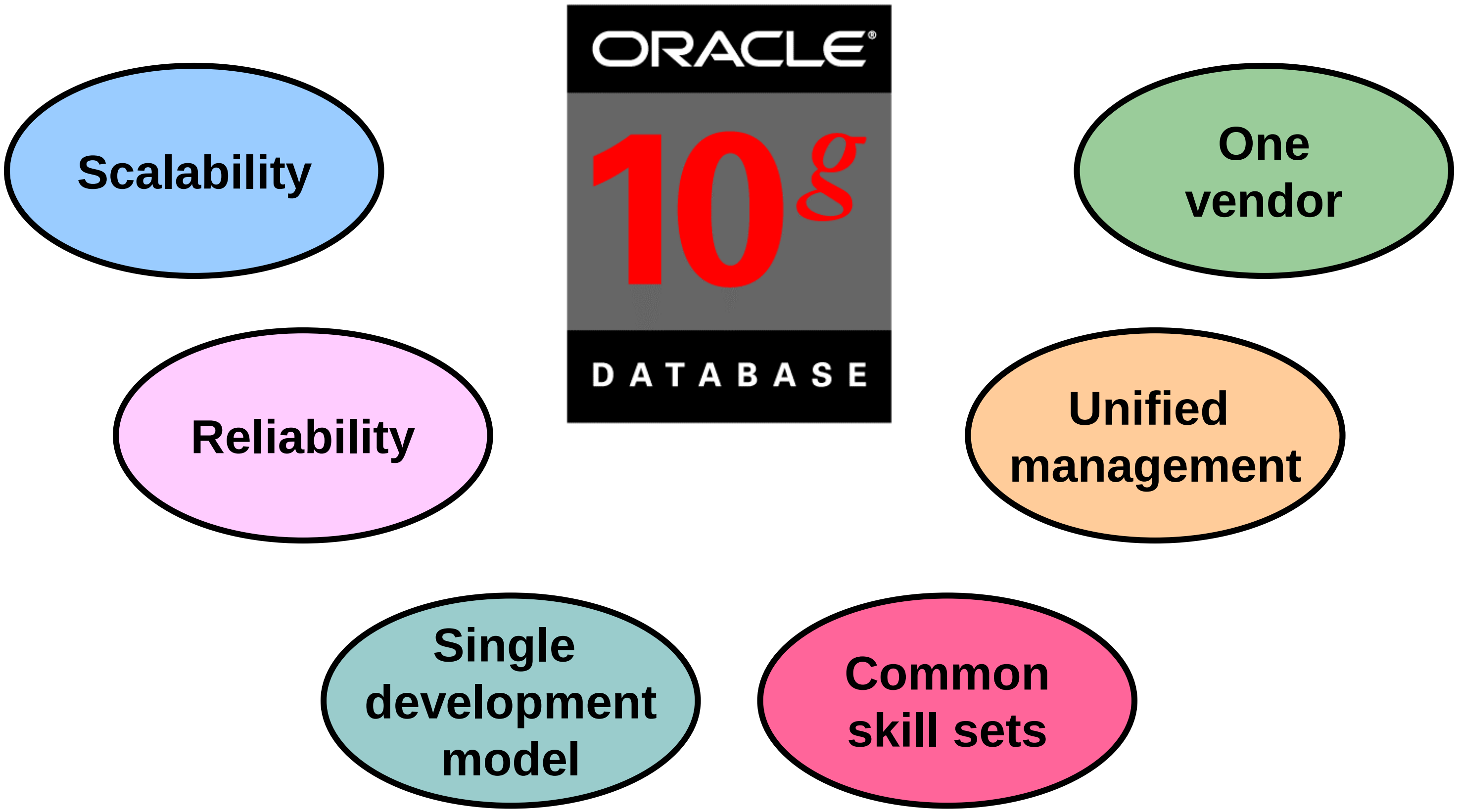
- List the features of Oracle10*g*
- Discuss the theoretical and physical aspects of a relational database
- Describe the Oracle implementation of the RDBMS and ORDBMS
- Understand the goals of the course

Goals of the Course

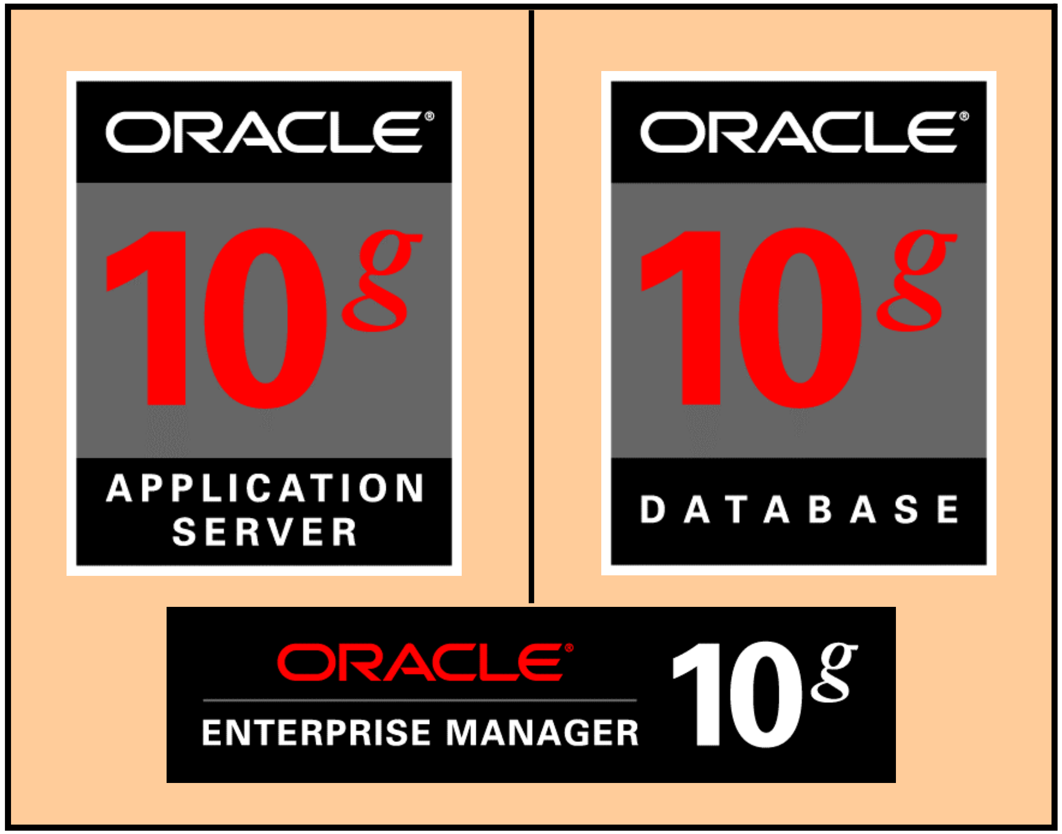
After completing this course, you should be able to do the following:

- Identify the major structural components of Oracle Database 10*g*
- 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 data manipulation language (DML) statements to update data in Oracle Database 10*g*
- Obtain metadata by querying the dictionary views

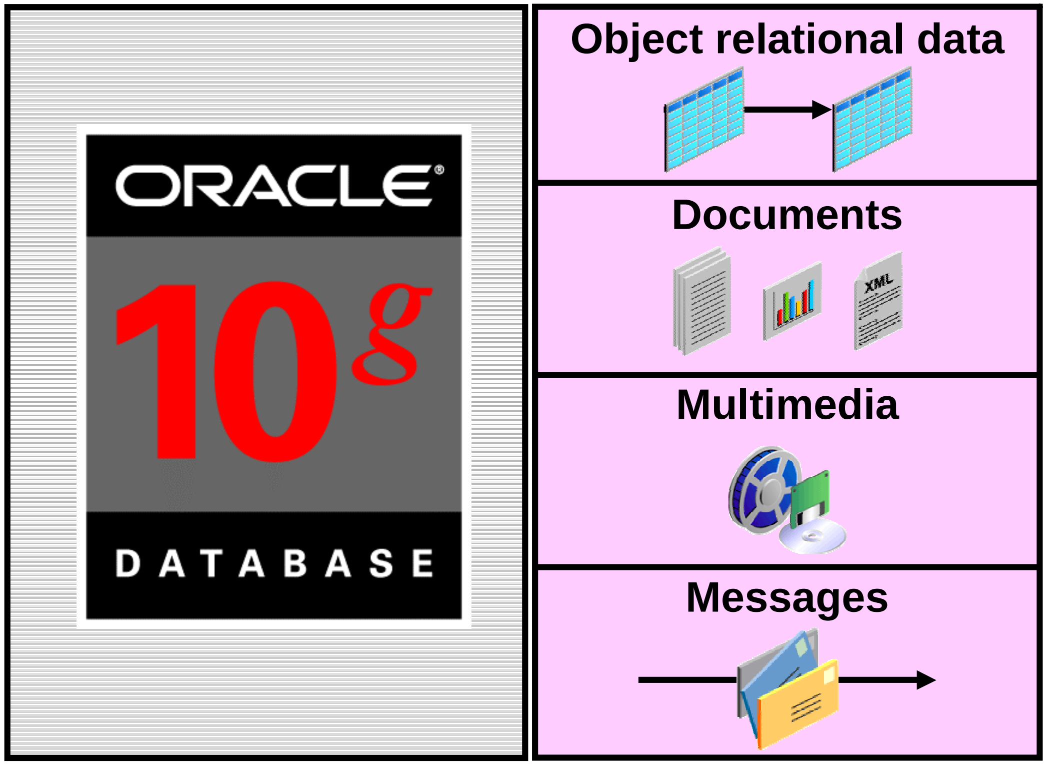
Oracle10*g*


ORACLE
10g
DATABASE
Scalability
Reliability
Single development model
Common skill sets
Unified management
One vendor

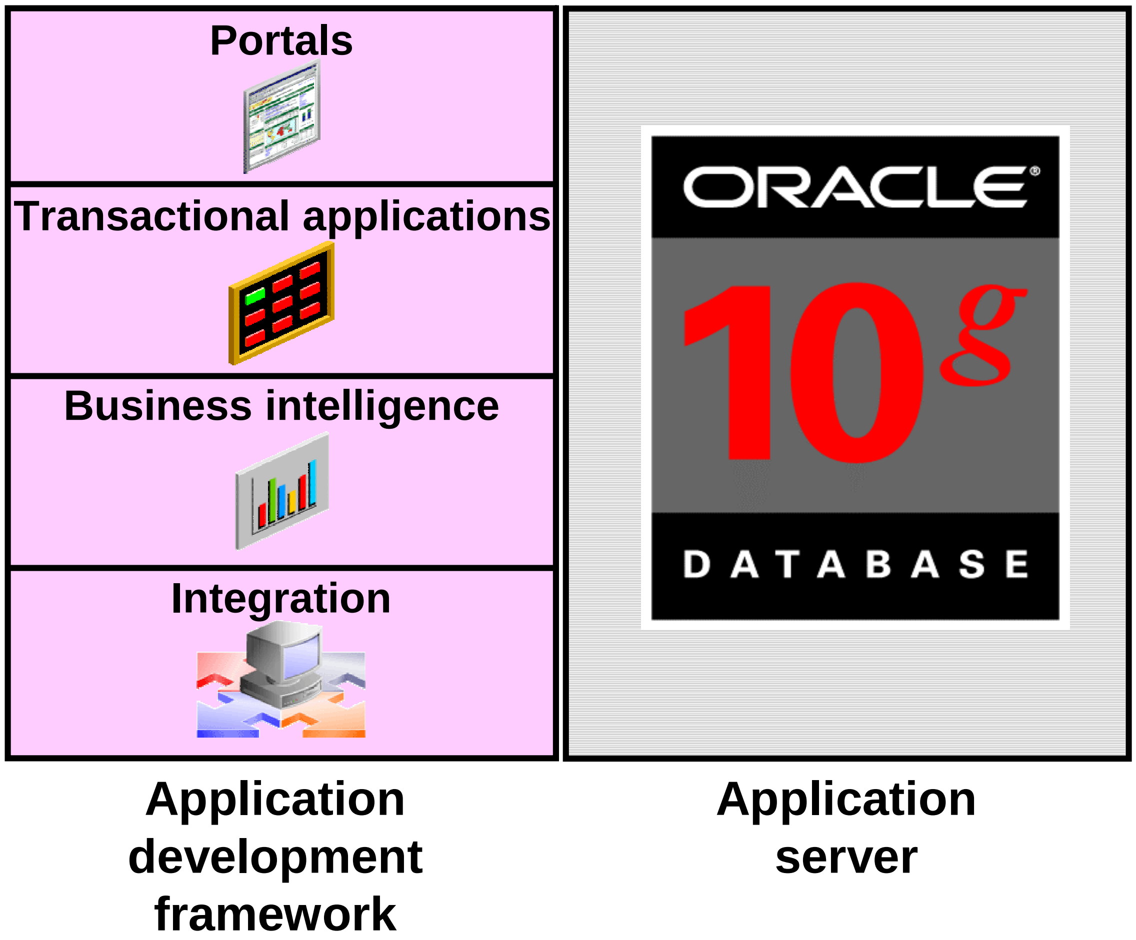
Oracle10*g*


ORACLE
10g
APPLICATION SERVER
ORACLE
10g
DATABASE
ORACLE
ENTERPRISE MANAGER
10g

Oracle Database 10*g*



Oracle Application Server 10*g*



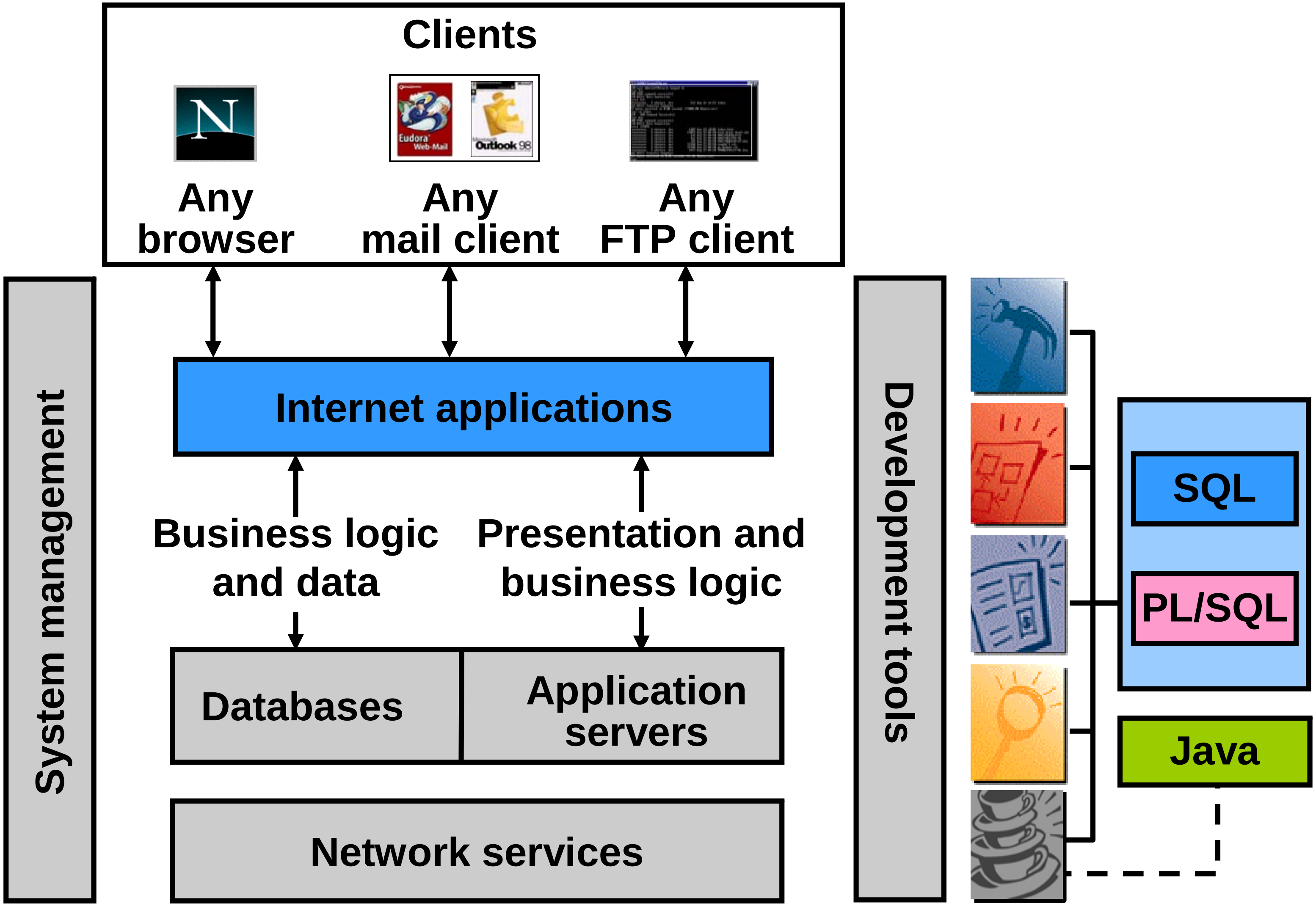
Oracle Enterprise Manager 10*g* Grid Control

- Software provisioning
- Application service level monitoring

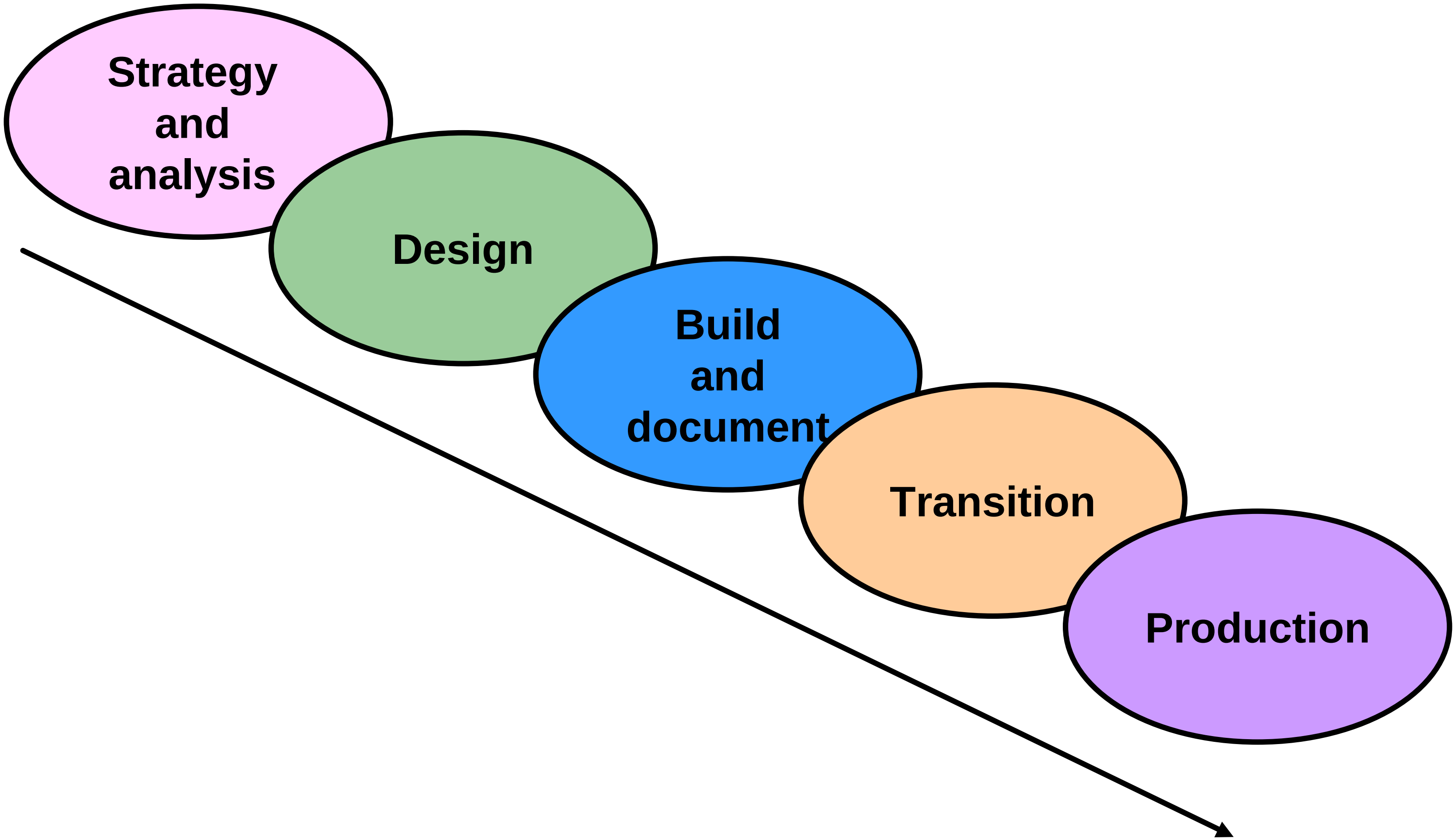
Relational and Object Relational Database Management Systems

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

Oracle Internet Platform



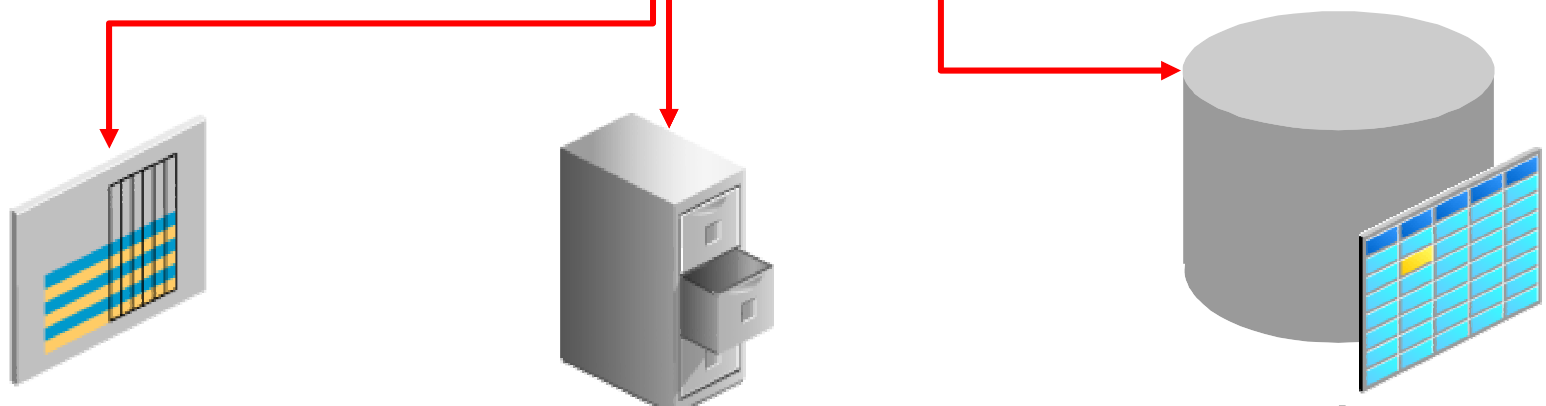
System Development Life Cycle



Data Storage on Different Media

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

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



Electronic spreadsheet

Filing cabinet

Database

ORACLE

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.

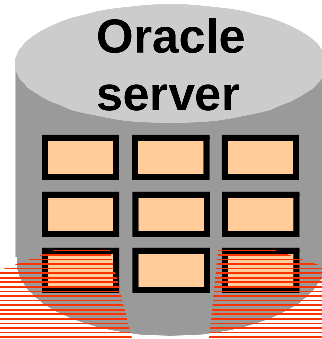


Table name: `EMPLOYEES`

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PH
100	Steven	King	SKING	51
101	Neena	Kochhar	NKOCHHAR	51
102	Lex	De Haan	LDEHAAN	51

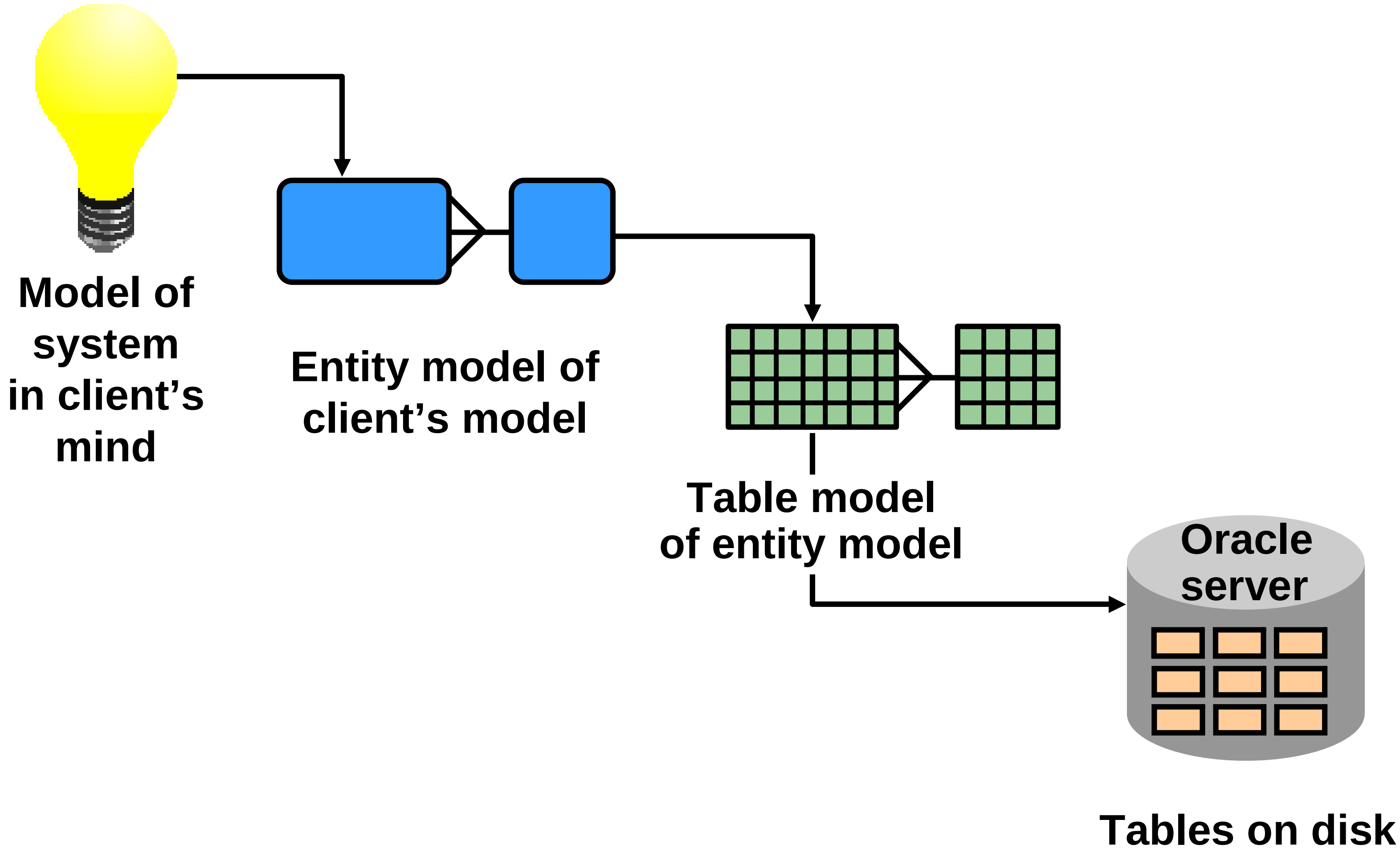
…

Table name: `DEPARTMENTS`

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID
10	Administration	200
20	Marketing	201
50	Shipping	124

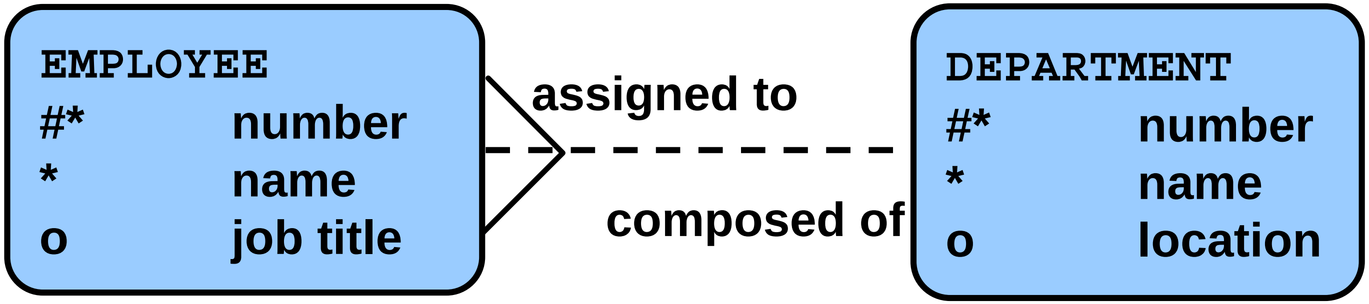
…

Data Models


Model of
system
in client's
mind
Entity model of
client's model
Table model
of entity model
Oracle
server
Tables on disk

Entity Relationship Model

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

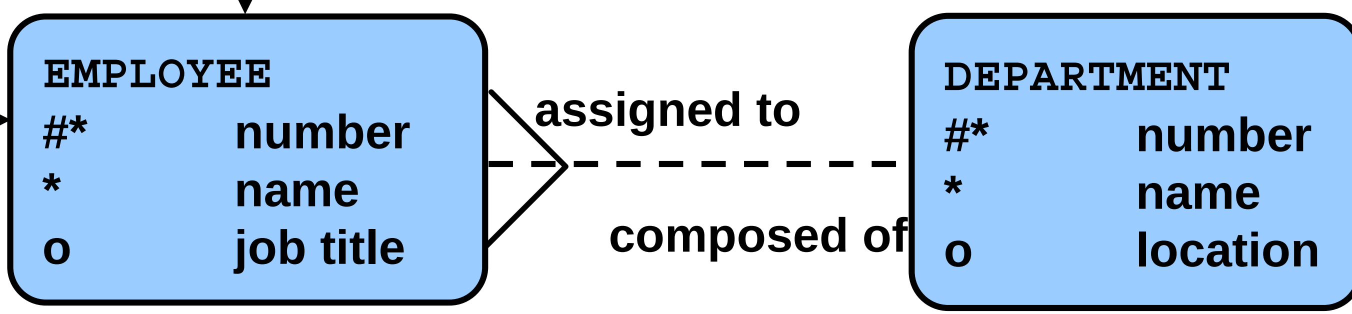


- Scenario
  - "... Assign one or more employees to a department ..."
  - "... Some departments do not yet have assigned employees ..."

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"



Unique identifier (UID)
Primary marked with "#"
Secondary marked with "(#)"

Relating Multiple Tables

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

Table name: `EMPLOYEES`

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID
174	Ellen	Abel	80
142	Curtis	Davies	50
102	Lex	De Haan	90
104	Bruce	Ernst	60
202	Pat	Fay	20
206	William	Gietz	110

...

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		1700

Primary key **Foreign key** **Primary key**

Relational Database Terminology

EMPLOYEE_ID	LAST_NAME	FIRST_NAME	SALARY	COMMISSION_PCT	DEPARTMENT_ID
100	King	Steven	24000		90
101	Kochhar	Neena	17000		90
102	De Haan	Lex	17000		90
103	Hunold	Alexander	9000		60
104	Ernst	Bruce	6000		60
107	Lorentz	Diana	4200		60
124	Mourgos	Kevin	5800		50
141	Rajs	Trenna	3500		50
142	Davies	Curtis	3100		50
143	Matos	Randall	2600		50
144	Vargas	Peter	2500		50
149	Zlotkey	Eleni	10500	.2	80
174	Abel	Ellen	11000	.3	80
176	Taylor	Jonathon	8600	.2	80
178	Grant	Kimberely	7000	.15	
200	Whalen	Jennifer	4400		10
201	Hartstein	Michael	13000		20
202	Fay	Pat	6000		20
205	Higgins	Shelley	12000		110
206	Gietz	William	8300		110

Relational Database Properties

A relational database:

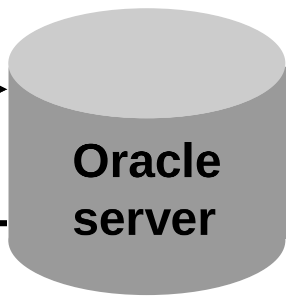
- Can be accessed and modified by executing structured query language (SQL) statements
- Contains a collection of tables with no physical pointers
- Uses a set of operators

Communicating with an RDBMS Using SQL

SQL statement is entered.

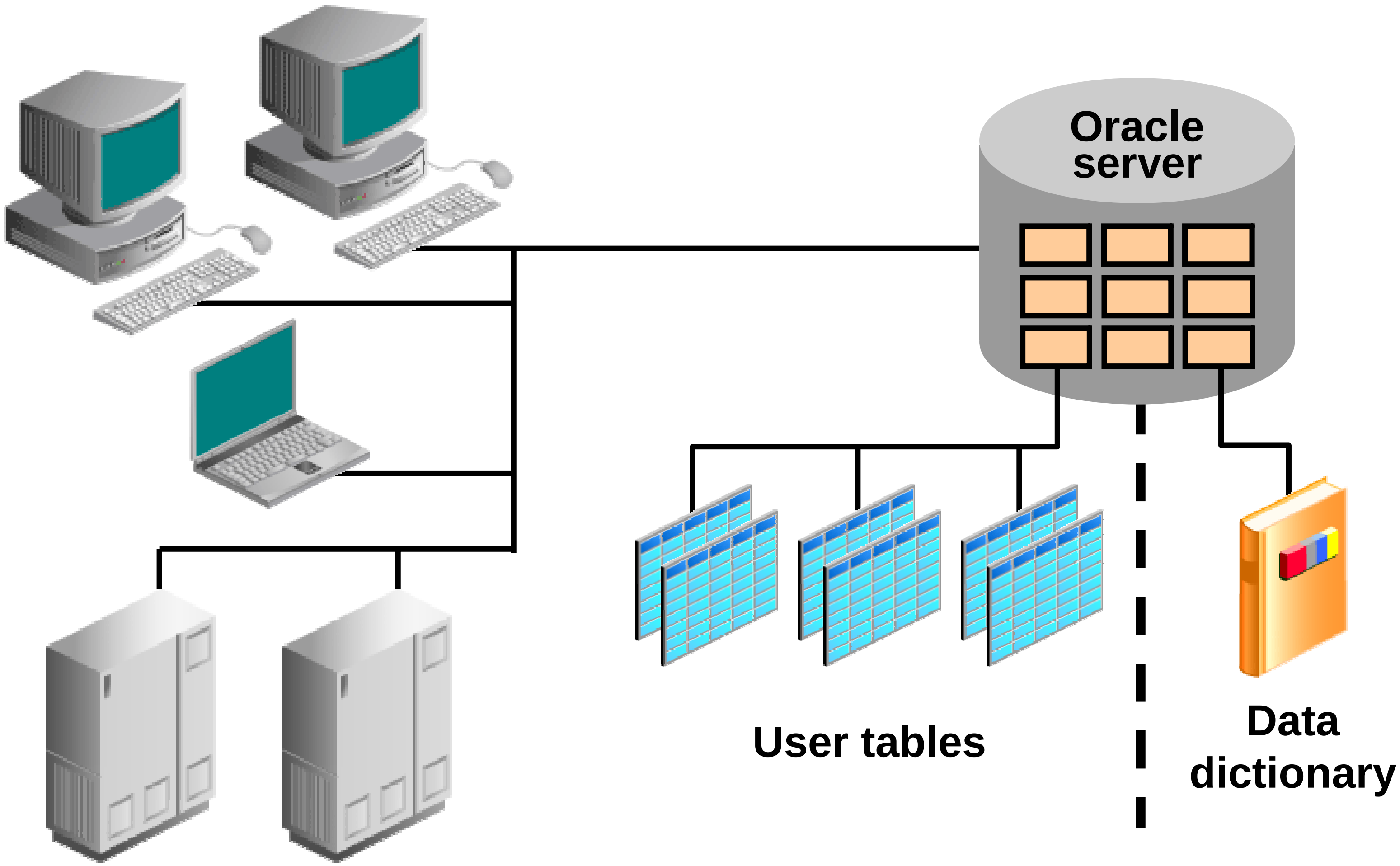
```
SELECT department_name
FROM   departments;
```

Statement is sent to Oracle server.



DEPARTMENT_NAME
Administration
Marketing
Shipping
IT
Sales
Executive
Accounting
Contracting

Oracle's Relational Database Management System



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

Tables Used in the Course

EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALA
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	240
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	170
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	170
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	90
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	60
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	42
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	58
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	35
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	31
				1.2874	15-MAR-98	ST_CLERK	26
				1.2004	09-JUL-98	ST_CLERK	25

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		1700

DEPARTMENTS

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

JOB_GRADES

Summary

- Oracle Database 10*g* is the database for grid computing.
- The database is based on the object relational database management system.
- 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 the SQL language and PL/SQL engine.

1

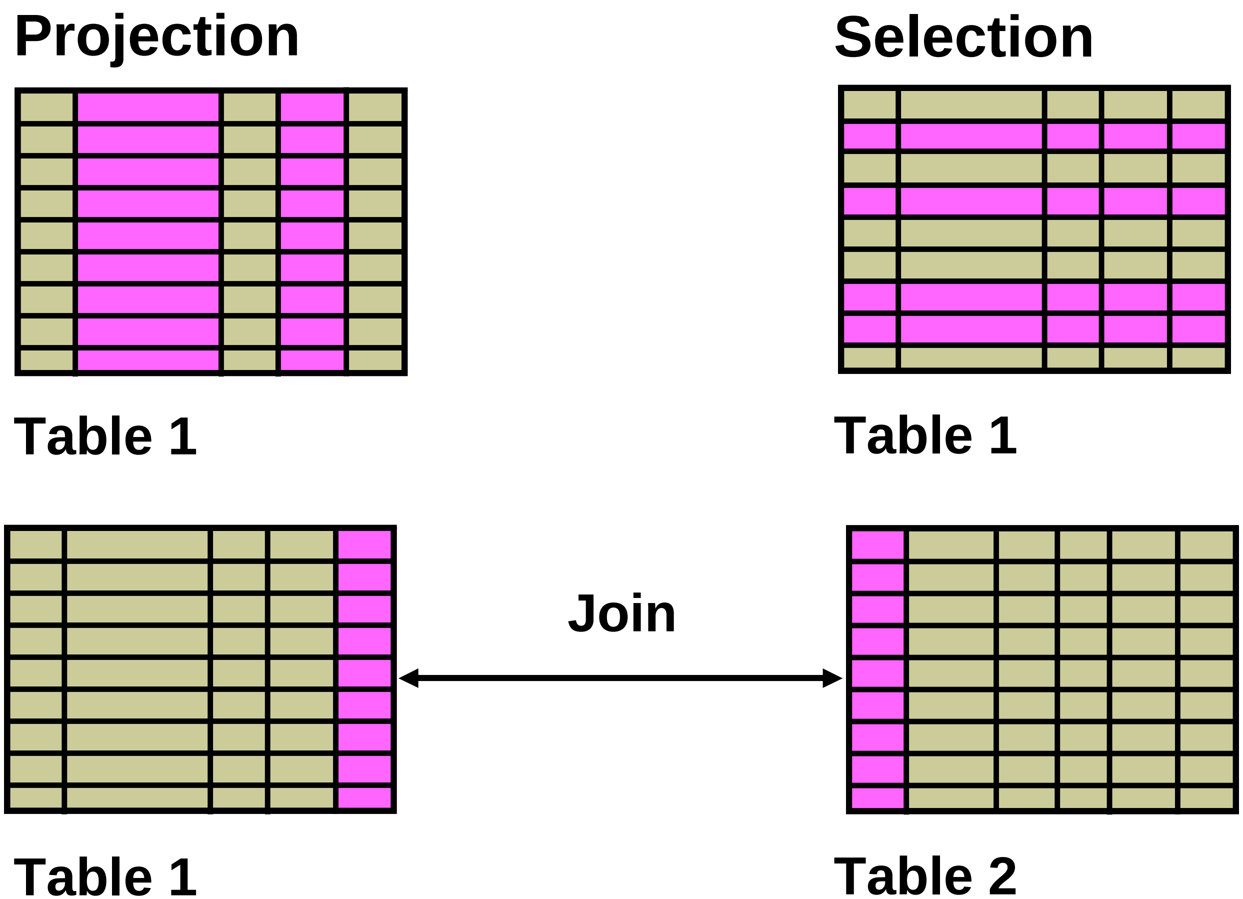
Retrieving Data Using the SQL SELECT Statement

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**
- **Differentiate between SQL statements and *i*SQL*Plus commands**

Capabilities of SQL `SELECT` Statements



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
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		1700

8 rows selected.

Selecting Specific Columns

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

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

8 rows selected.

Writing SQL Statements

- SQL statements are not case-sensitive.
- SQL statements can be 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 *i*SQL*Plus, SQL statements can optionally be terminated by a semicolon (;). Semicolons are required if you execute multiple SQL statements.
- In SQL*plus, you are required to end each SQL statement with a semicolon (;).

Column Heading Defaults

- *i*SQL*Plus:
  - Default heading alignment: Center
  - Default heading display: Uppercase
- SQL*Plus:
  - Character and Date column headings are left-aligned
  - Number column headings are right-aligned
  - Default heading display: Uppercase

ORACLE

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
King	24000	24300
Kochhar	17000	17300
De Haan	17000	17300
Hunold	9000	9300
Ernst	6000	6300

...

20 rows selected.

Operator Precedence

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

1

LAST_NAME	SALARY	12*SALARY+100
King	24000	288100
Kochhar	17000	204100
De Haan	17000	204100

...

20 rows selected.

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

2

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

...

20 rows selected.

Defining a Null Value

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

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

LAST_NAME	JOB_ID	SALARY	COMMISSION_PCT
King	AD_PRES	24000	
Kochhar	AD_VP	17000	
...			
Zlotkey	SA_MAN	10500	.2
Abel	SA_REP	11000	.3
Taylor	SA_REP	8600	.2
...			
Gietz	AC_ACCOUNT	8300	

20 rows selected.

Null Values in Arithmetic Expressions

Arithmetic expressions containing a null value evaluate to null.

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

LAST_NAME	12*SALARY*COMMISSION_PCT
King	
Kochhar	

...

Zlotkey	25200
Abel	39600
Taylor	20640

...

Gietz	

20 rows selected.

ORACLE

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
King	
Kochhar	
De Haan	

...

20 rows selected.

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

Name	Annual Salary
King	288000
Kochhar	204000
De Haan	204000

...

20 rows selected.

ORACLE

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;
```

Employees
KingAD_PRES
KochharAD_VP
De HaanAD_VP

...

20 rows selected.

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 by single quotation marks.
- Each character string is output once for each row returned.

Using Literal Character Strings

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

Employee Details
King is a AD_PRES
Kochhar is a AD_VP
De Haan is a AD_VP
Hunold is a IT_PROG
Ernst is a IT_PROG
Lorentz is a IT_PROG
Mourgos is a ST_MAN
Rajs is a ST_CLERK

...

20 rows selected.

Alternative Quote (q) Operator

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

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

Department and Manager
Administration, it's assigned manager ID: 200
Marketing, it's assigned manager ID: 201
Shipping, it's assigned manager ID: 124

...

8 rows selected.

Duplicate Rows

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

```
SELECT department_id
FROM   employees;
```

1

DEPARTMENT_ID
90
90
90

…

20 rows selected.

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

2

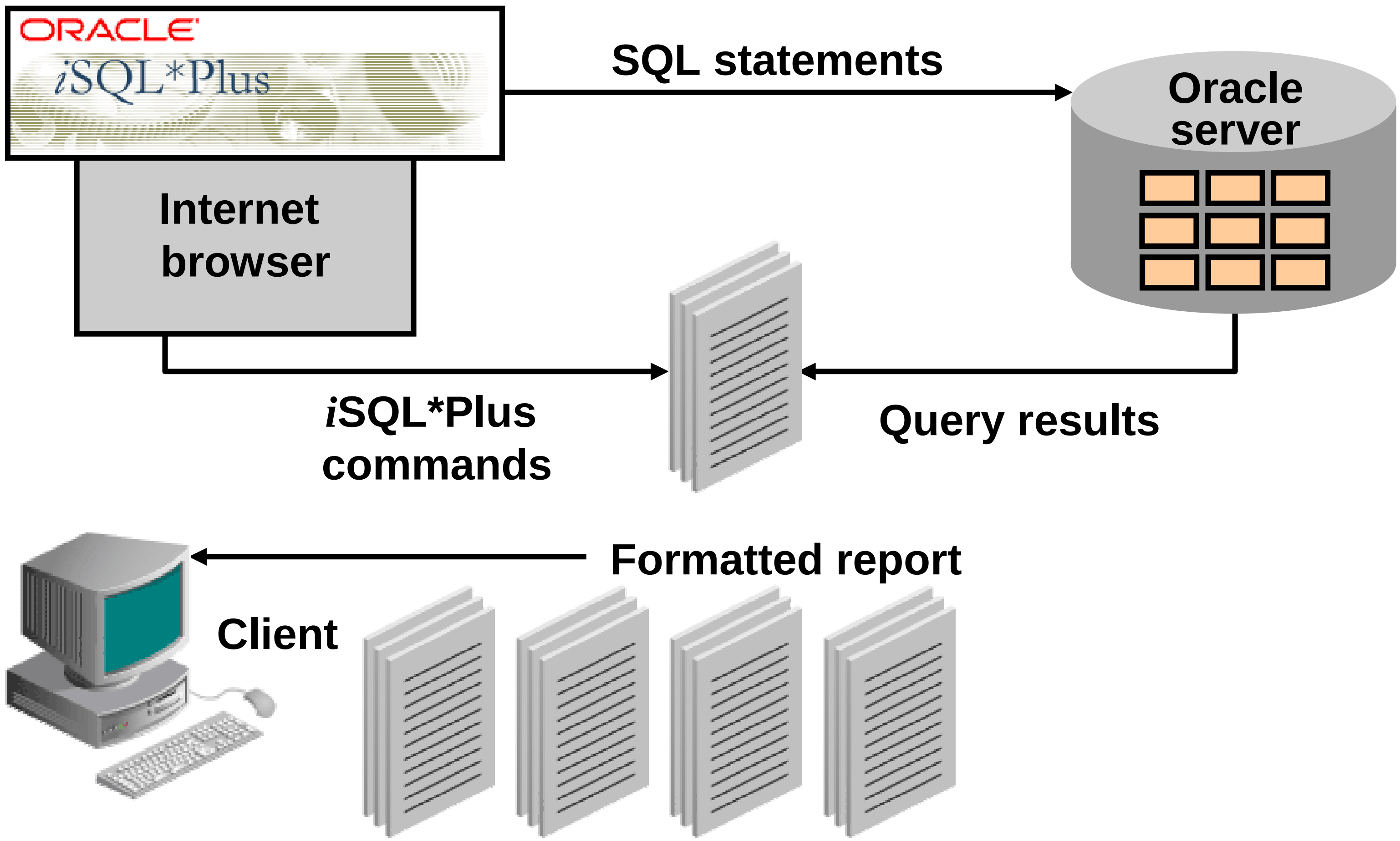
DEPARTMENT_ID
10
20
50

…

8 rows selected.

ORACLE

SQL and *i*SQL*Plus Interaction



SQL Statements Versus *i*SQL*Plus Commands

SQL

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

SQL statements

***i*SQL*Plus**

- An environment
- Oracle-proprietary
- Keywords can be abbreviated
- Commands do not allow manipulation of values in the database
- Runs on a browser
- Centrally loaded; does not have to be implemented on each machine

***i*SQL*Plus commands**

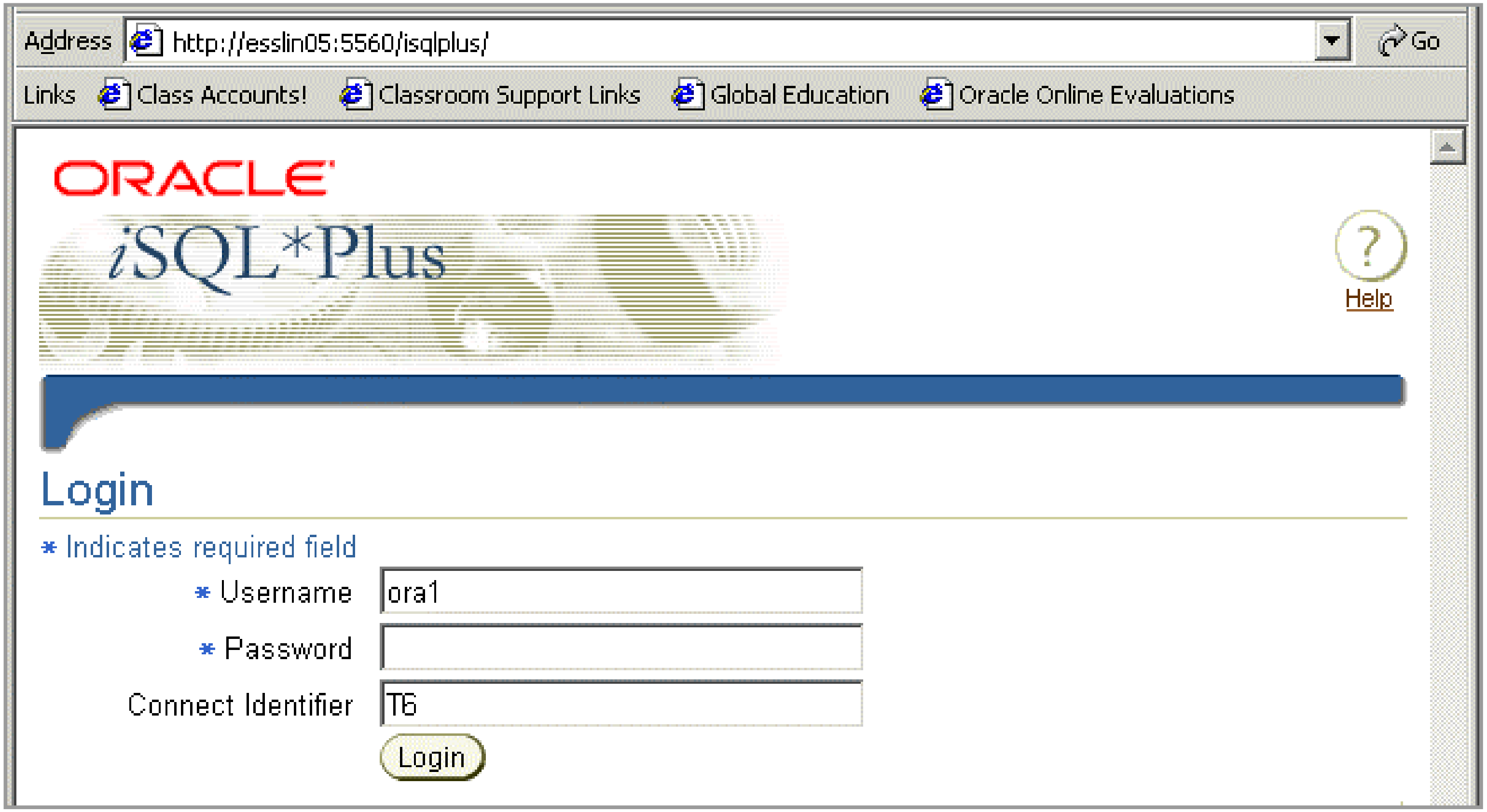
Overview of *i*SQL*Plus

After you log in to *i*SQL*Plus, you can:

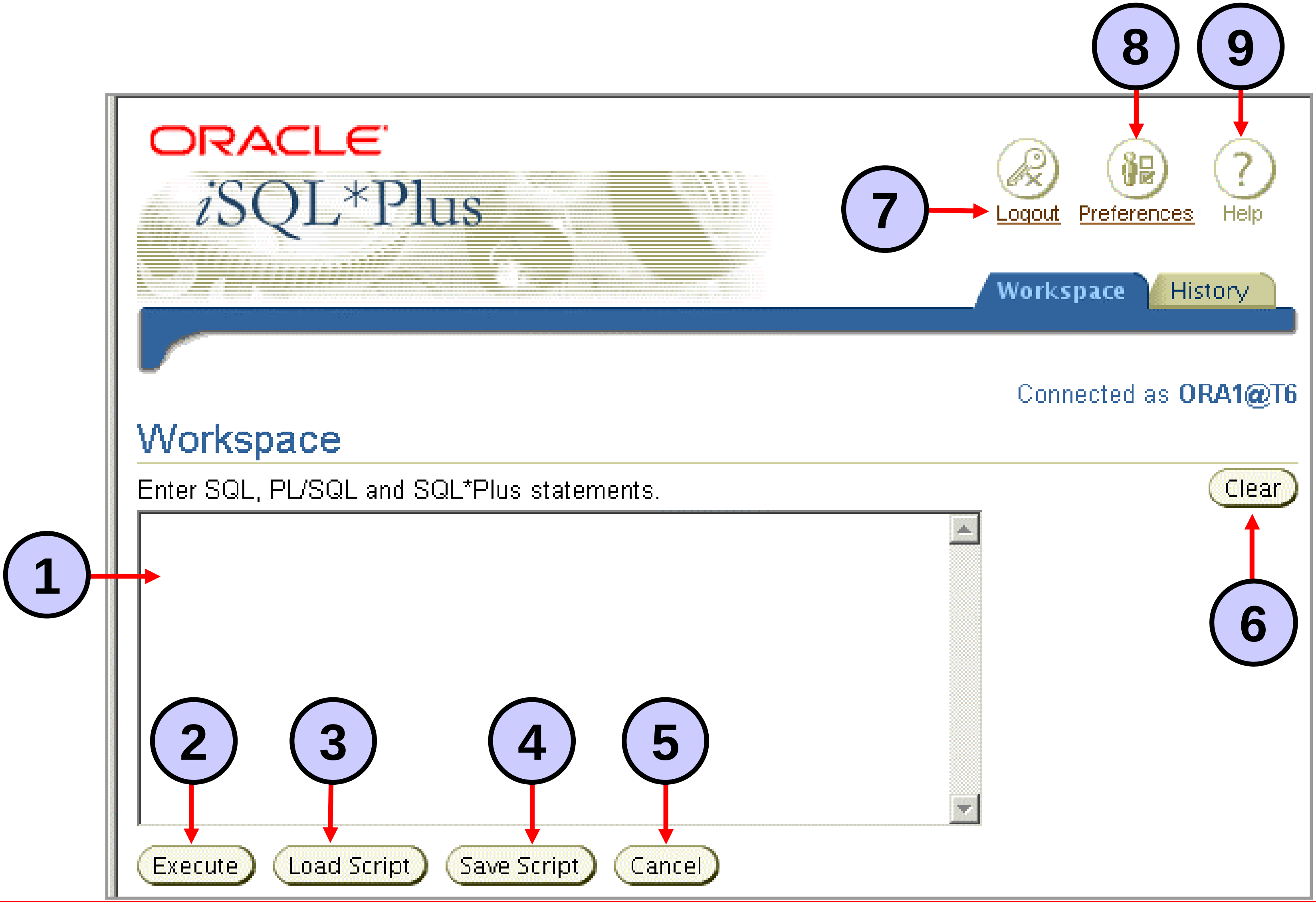
- **Describe table structures**
- **Enter, execute, and edit SQL statements**
- **Save or append SQL statements to files**
- **Execute or edit statements that are stored in saved script files**

Logging In to *i*SQL*Plus

From your browser environment:



*i*SQL*Plus Environment



Displaying Table Structure

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

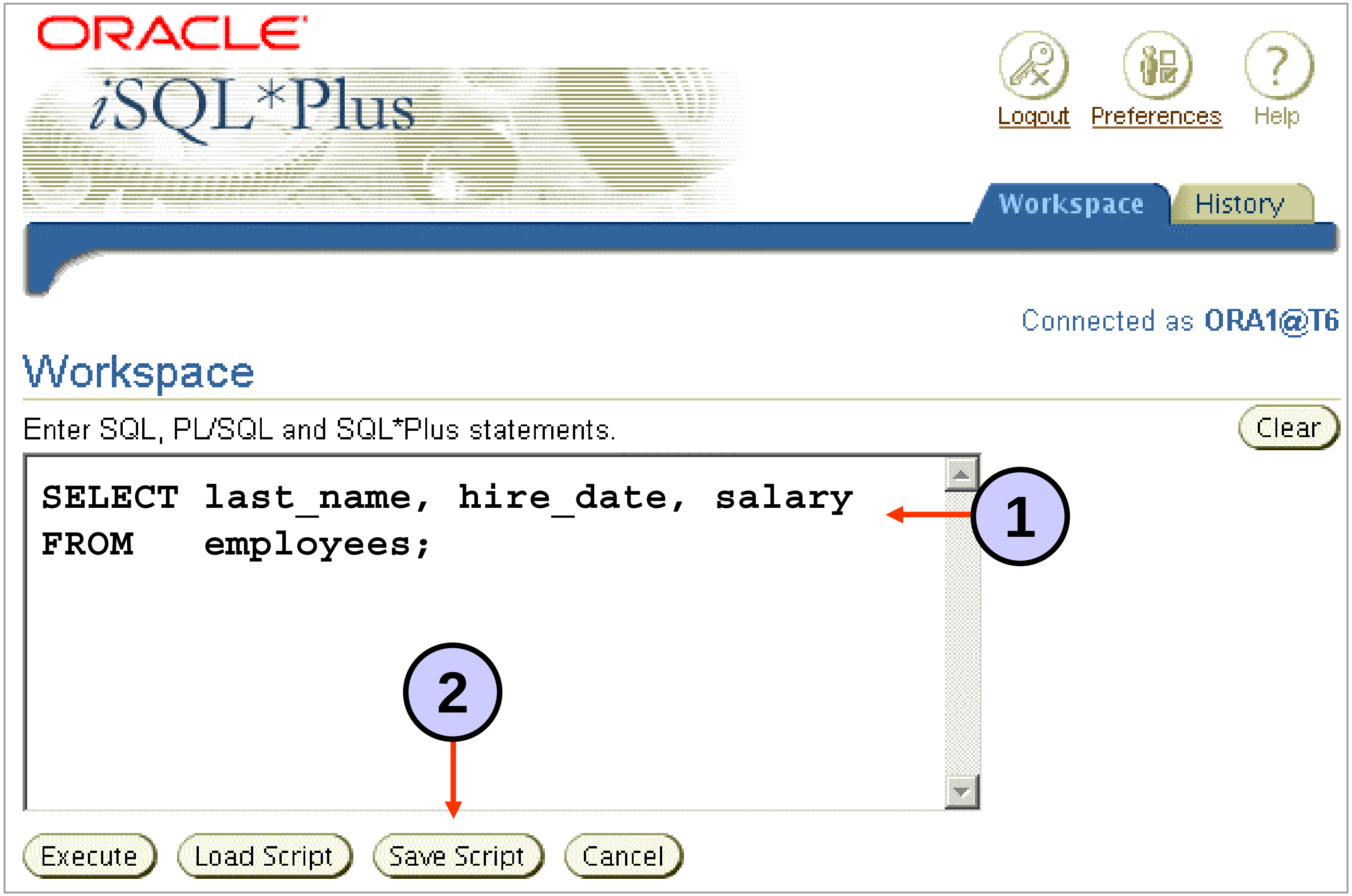
```
DESC[RIBE] tablename
```

Displaying Table Structure

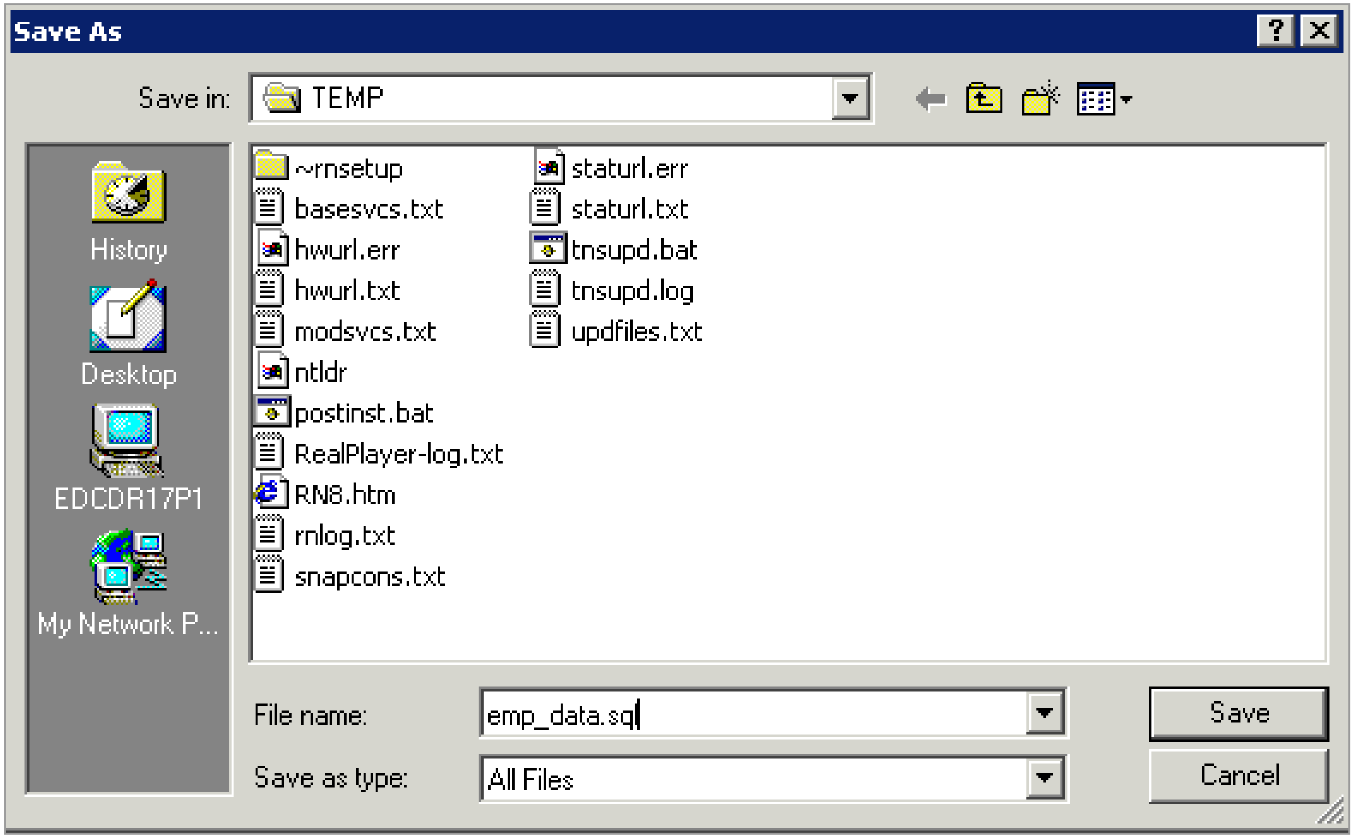
```
DESCRIBE employees
```

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)

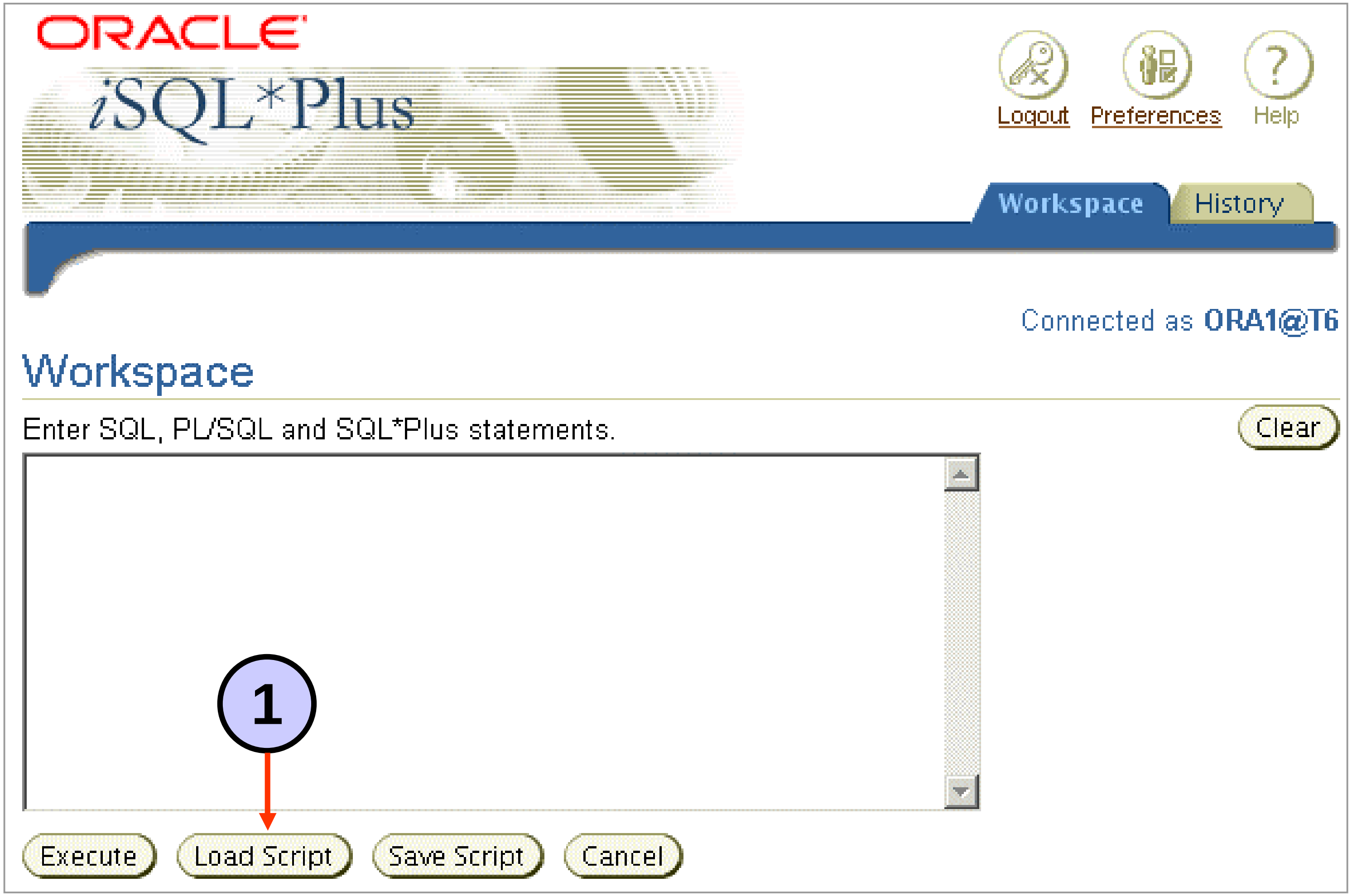
Interacting with Script Files



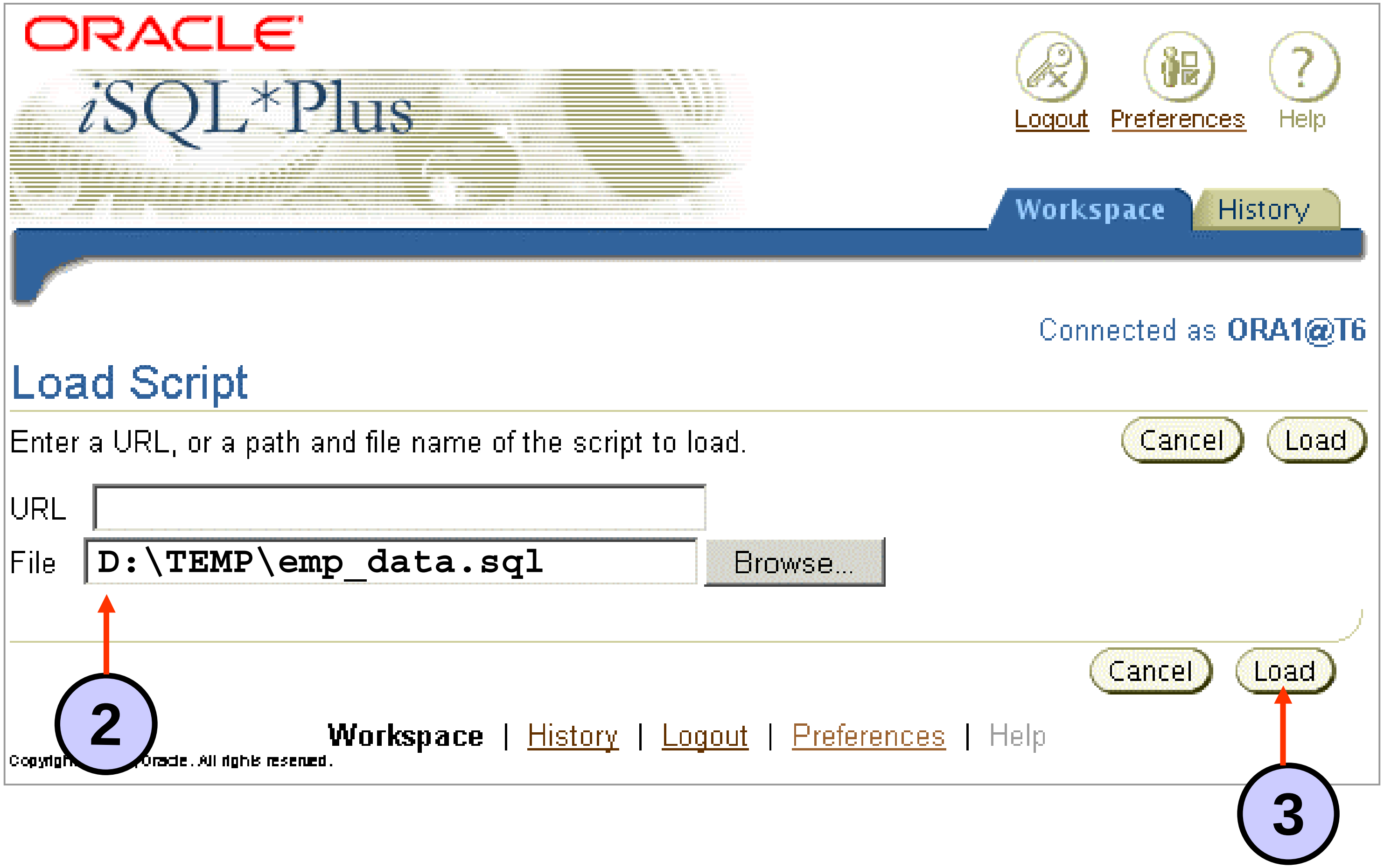
Interacting with Script Files



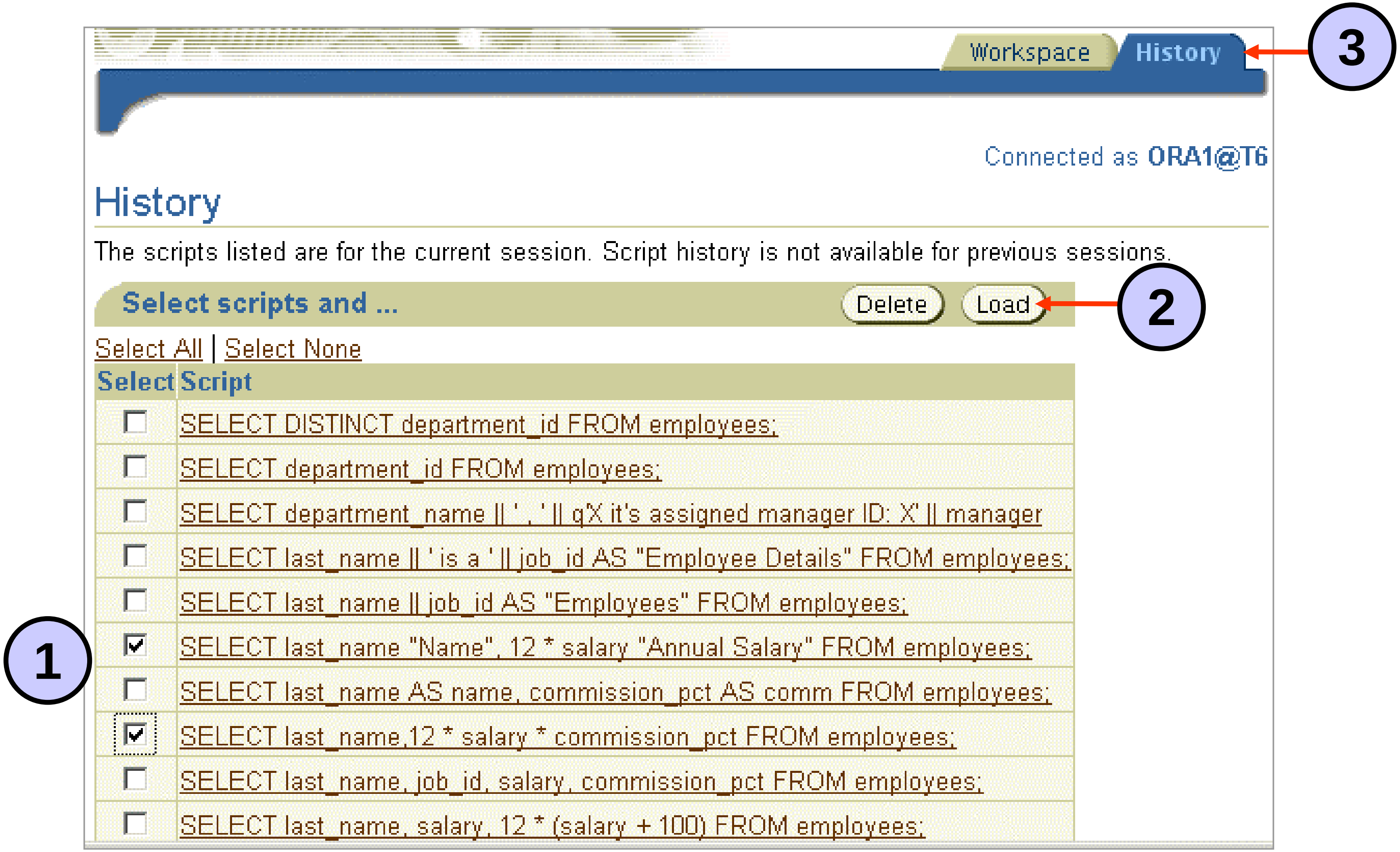
Interacting with Script Files



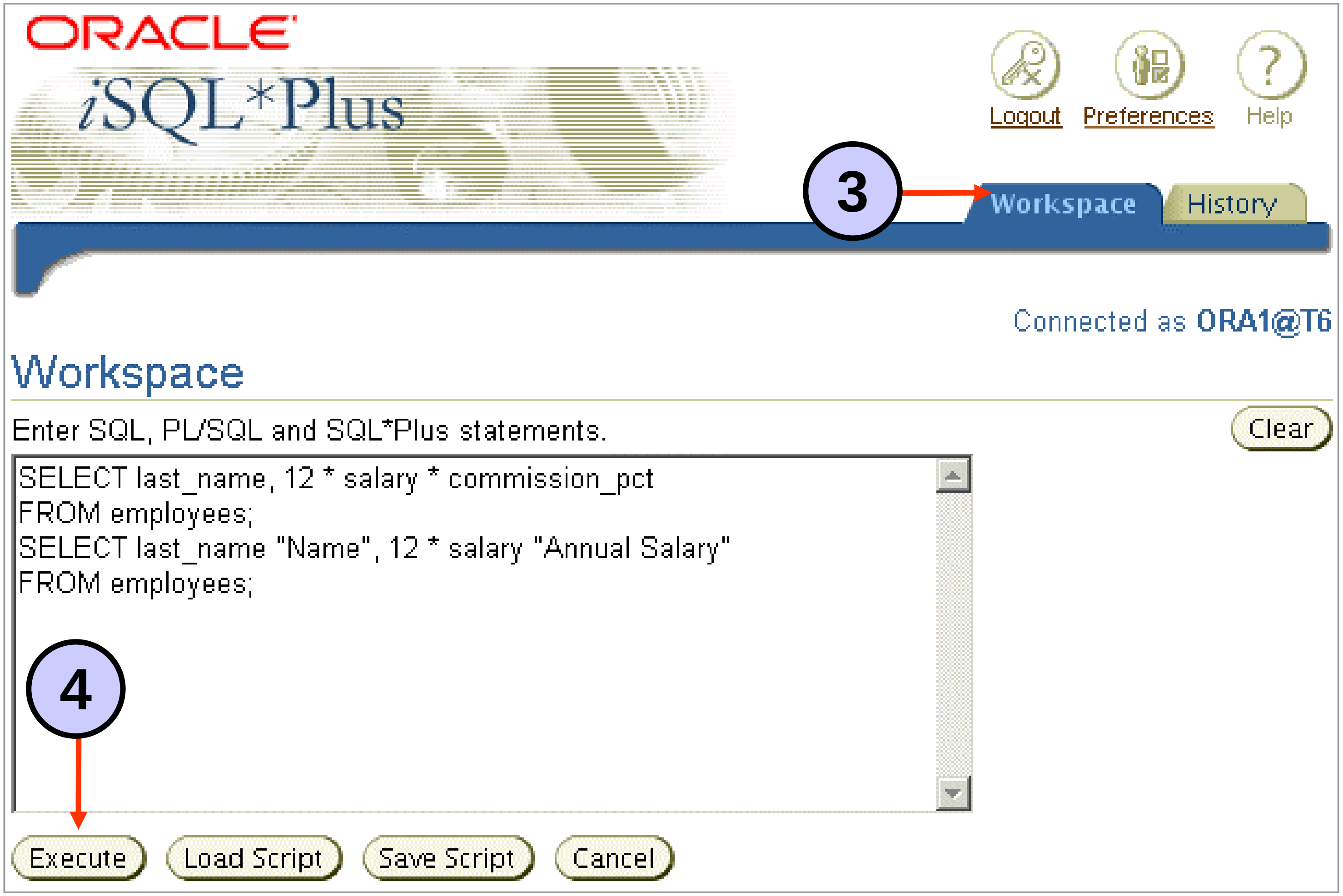
Interacting with Script Files



*i*SQL*Plus History Page

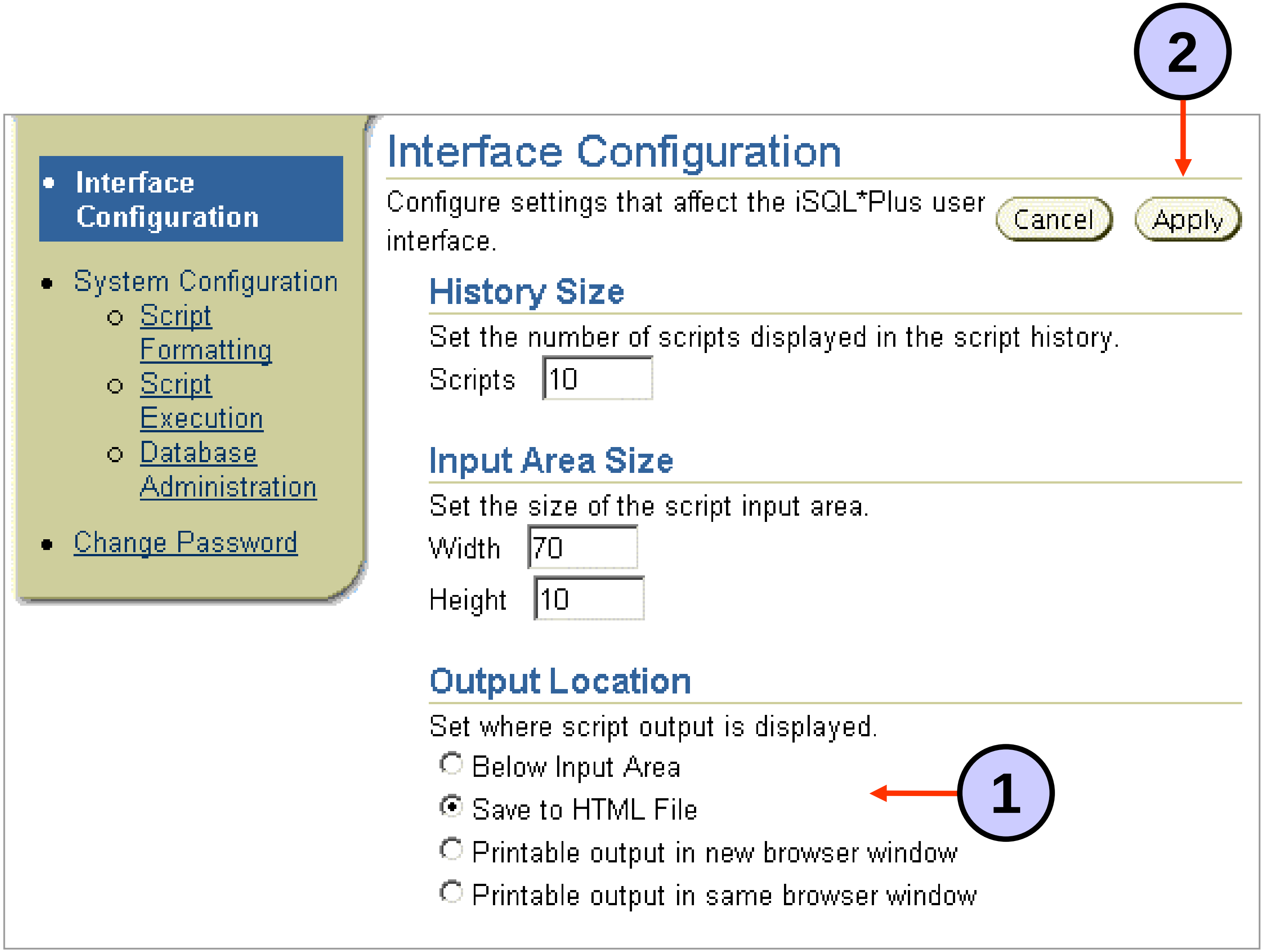


*i*SQL*Plus History Page



Setting *i*SQL*Plus Preferences

Setting the Output Location Preference



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
- Use the *i*SQL*Plus environment to write, save, and execute SQL statements and *i*SQL*Plus commands

```
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
- Using *i*SQL*Plus

2

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 in *i*SQL*Plus to restrict and sort output at run time**

Limiting Rows Using a Selection

EMPLOYEES

EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90
103	Hunold	IT_PROG	60
104	Ernst	IT_PROG	60
107	Lorentz	IT_PROG	60
124	Mourgos	ST_MAN	50

…

20 rows selected.

“retrieve all employees in department 90”

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

Character Strings and Dates

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

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

Comparison Conditions

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

ORACLE

Using Comparison Conditions

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

LAST_NAME	SALARY
Matos	2600
Vargas	2500

ORACLE

Using the BETWEEN Condition

Use the BETWEEN condition 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
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500

Using the IN Condition

Use the IN membership condition 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
202	Fay	6000	201
200	Whalen	4400	101
205	Higgins	12000	101
101	Kochhar	17000	100
102	De Haan	17000	100
124	Mourgos	5800	100
149	Zlotkey	10500	100
201	Hartstein	13000	100

8 rows selected.

Using the `LIKE` Condition

- Use the `LIKE` condition 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%' ;
```

Using the LIKE Condition

- You can combine pattern-matching characters:

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

LAST_NAME
Kochhar
Lorentz
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
King	

Logical Conditions

Operator	Meaning
`AND`	Returns `TRUE` if *both* component conditions are true
`OR`	Returns `TRUE` if *either* component condition is true
`NOT`	Returns `TRUE` if the following condition is false

Using the AND Operator

AND requires both 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
149	Zlotkey	SA_MAN	10500
201	Hartstein	MK_MAN	13000

Using the `OR` Operator

`OR` requires either 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
100	King	AD_PRES	24000
101	Kochhar	AD_VP	17000
102	De Haan	AD_VP	17000
124	Mourgos	ST_MAN	5800
149	Zlotkey	SA_MAN	10500
174	Abel	SA_REP	11000
201	Hartstein	MK_MAN	13000
205	Higgins	AC_MGR	12000

8 rows selected.

ORACLE

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
King	AD_PRES
Kochhar	AD_VP
De Haan	AD_VP
Mourgos	ST_MAN
Zlotkey	SA_MAN
Whalen	AD_ASST
Hartstein	MK_MAN
Fay	MK_REP
Higgins	AC_MGR
Gietz	AC_ACCOUNT

10 rows selected.

ORACLE

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
King	AD_PRES	24000
Abel	SA_REP	11000
Taylor	SA_REP	8600
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
King	AD_PRES	24000

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
King	AD_PRES	90	17-JUN-87
Whalen	AD_ASST	10	17-SEP-87
Kochhar	AD_VP	90	21-SEP-89
Hunold	IT_PROG	60	03-JAN-90
Ernst	IT_PROG	60	21-MAY-91

...

20 rows selected.

Sorting

- **Sorting in descending order:**

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

1

- **Sorting by column alias:**

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

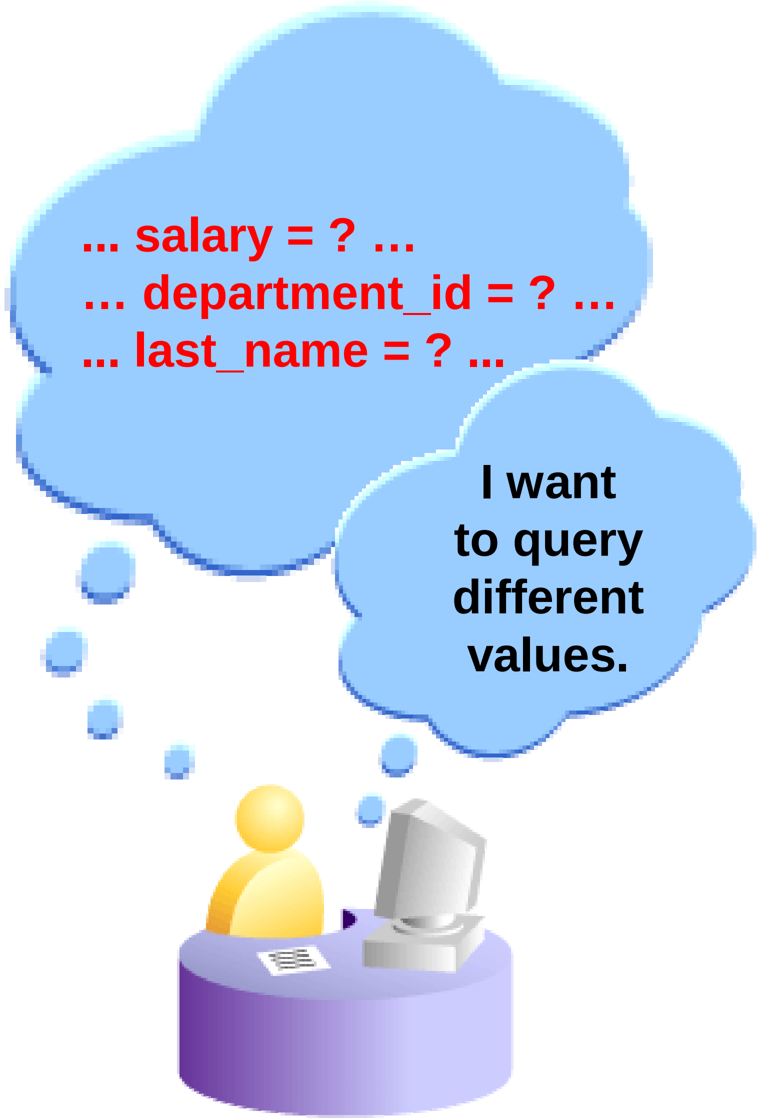
2

- **Sorting by multiple columns:**

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

3

Substitution Variables



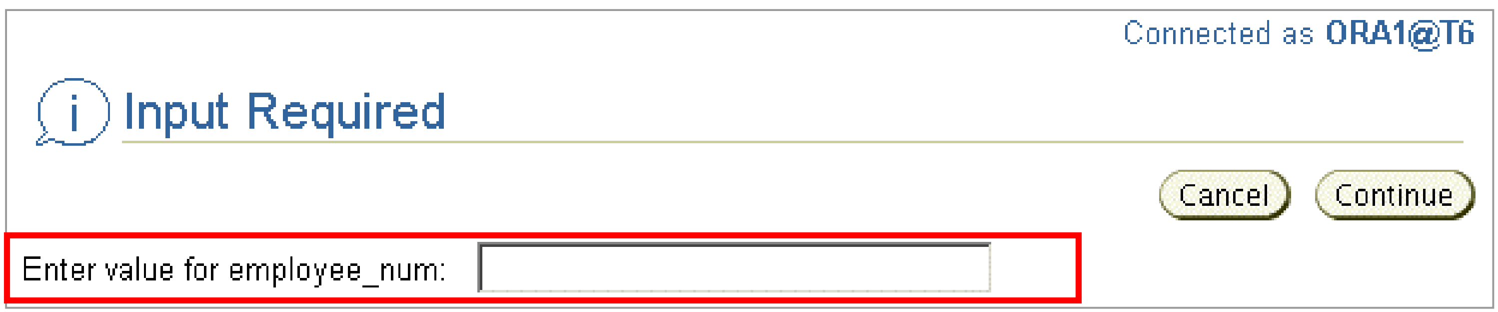
Substitution Variables

- Use *i*SQL*Plus 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

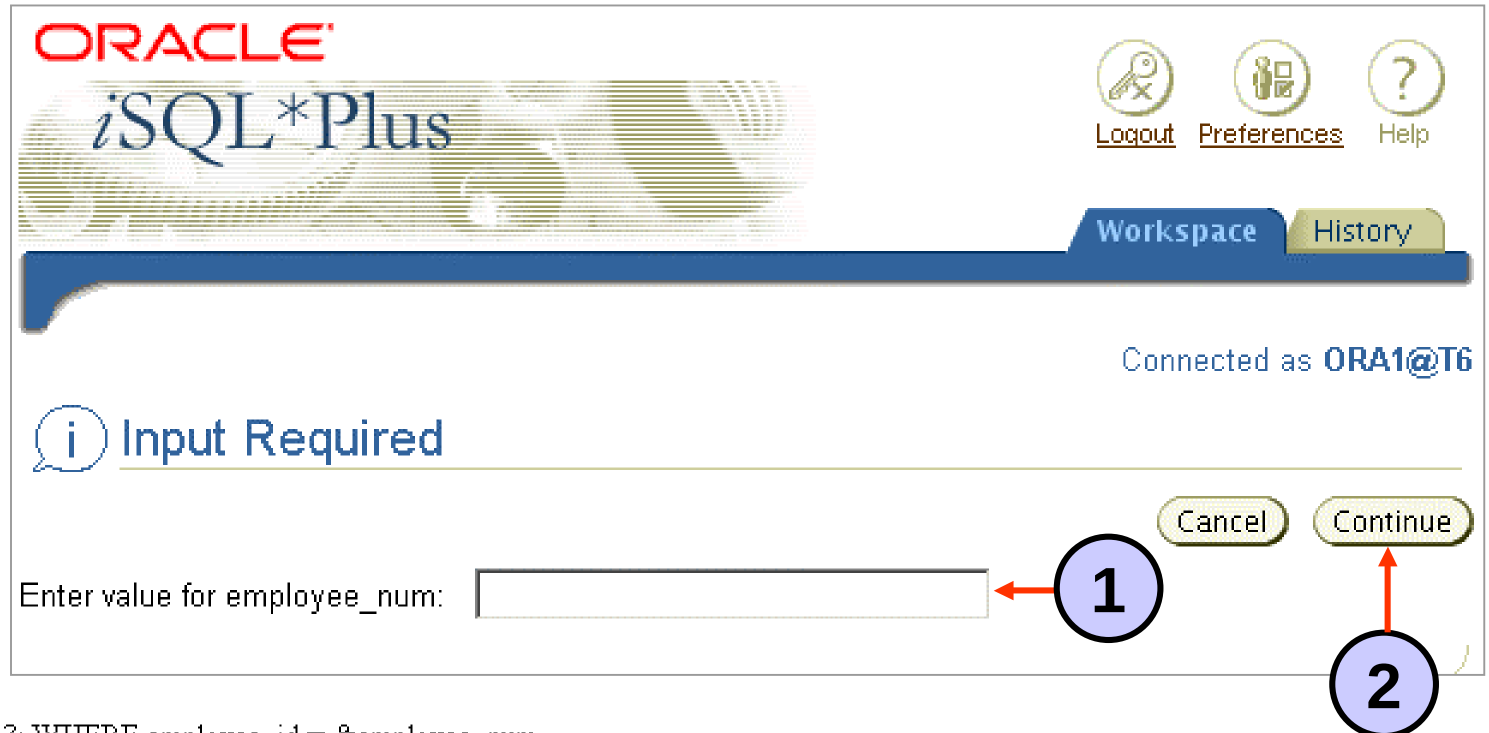
Using the & Substitution Variable

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 & Substitution Variable



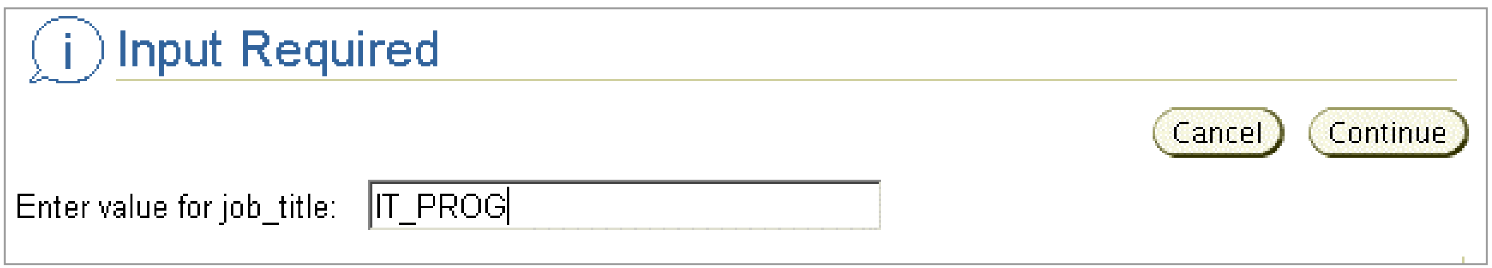
old 3: WHERE employee_id = &employee_num
new 3: WHERE employee_id = 101

EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID
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' ;
```



LAST_NAME	DEPARTMENT_ID	SALARY*12
Hunold	60	108000
Ernst	60	72000
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 ;
```

Input Required

Cancel Continue

Enter value for column_name: `salary`

Cancel Continue

Enter value for condition: `salary > 15000`

Cancel Continue

Enter value for order_column: `last_name`

ORACLE

Using the && Substitution Variable

Use the 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 ;
```

Input Required

Cancel Continue

Enter value for column_name: department_Id

EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
200	Whalen	AD_ASST	10
201	Hartstein	MK_MAN	20

...

20 rows selected.

Using the *i*SQL*Plus `DEFINE` Command

- Use the *i*SQL*Plus `DEFINE` command to create and assign a value to a variable.
- Use the *i*SQL*Plus `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
```

Using the `VERIFY` Command

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

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

"employee_num" 200

```
old   3: WHERE  employee_id = &employee_num
new   3: WHERE  employee_id = 200
```

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` conditions
  - 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 in *i*SQL*Plus 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

ORACLE

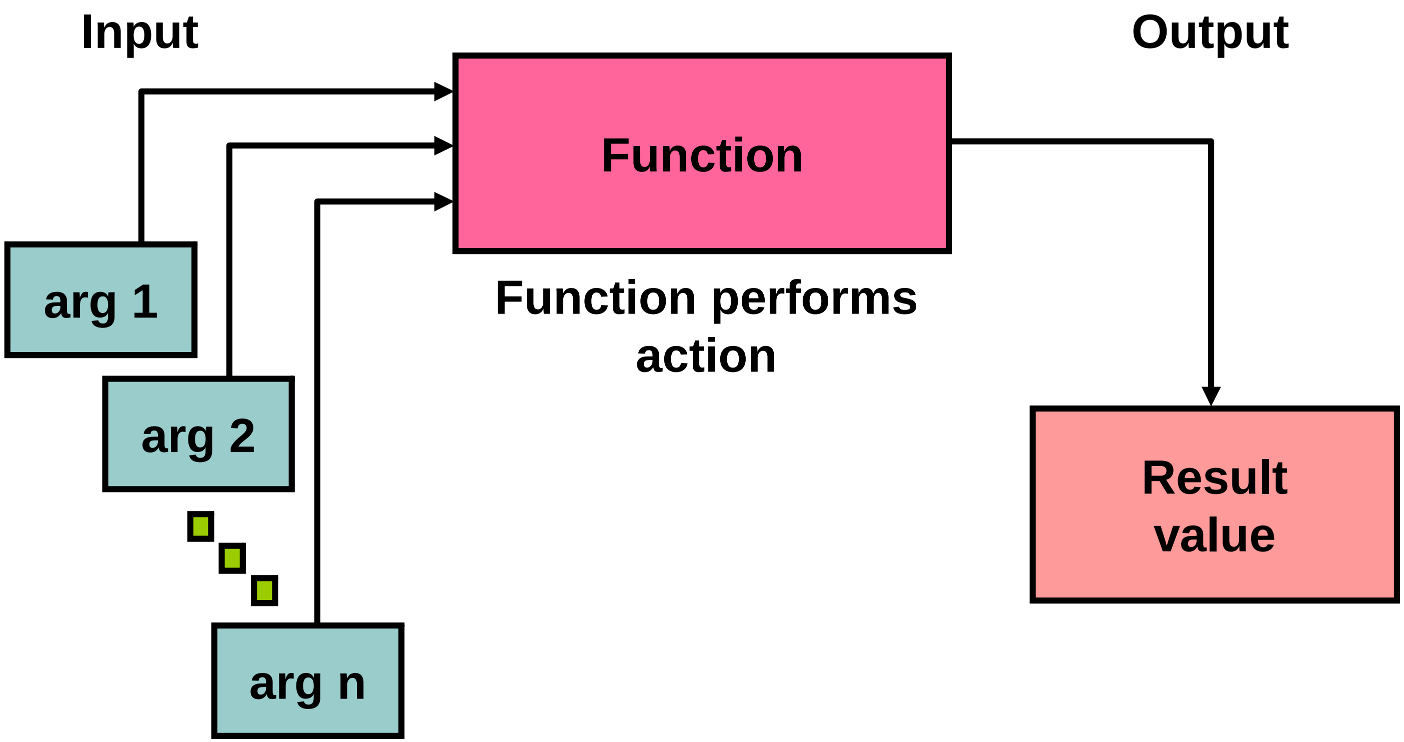
Using Single-Row Functions to Customize Output

Objectives

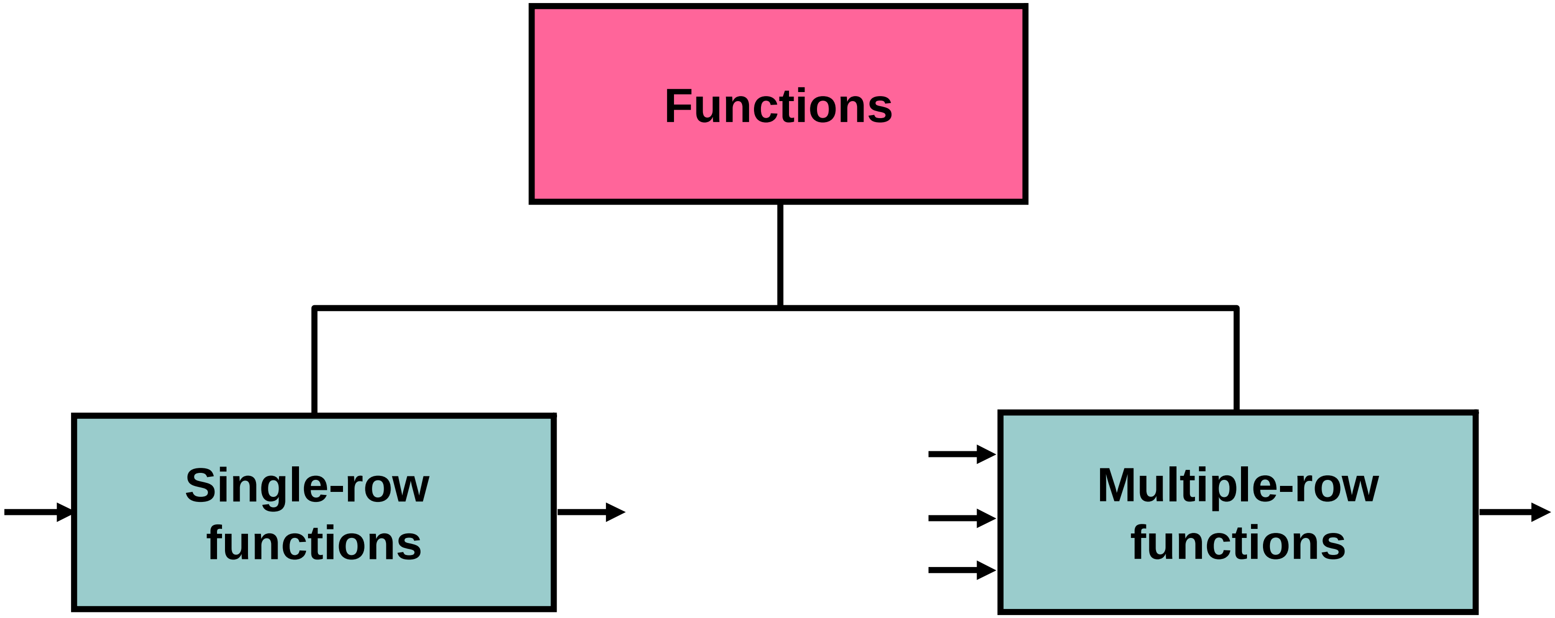
After completing this lesson, you should be able to do the following:

- Describe various types of functions that are available in SQL
- Use character, number, and date functions in `SELECT` statements
- Describe the use of conversion functions

SQL Functions



Two Types of SQL Functions



Return one result per row

Return one result per set of rows

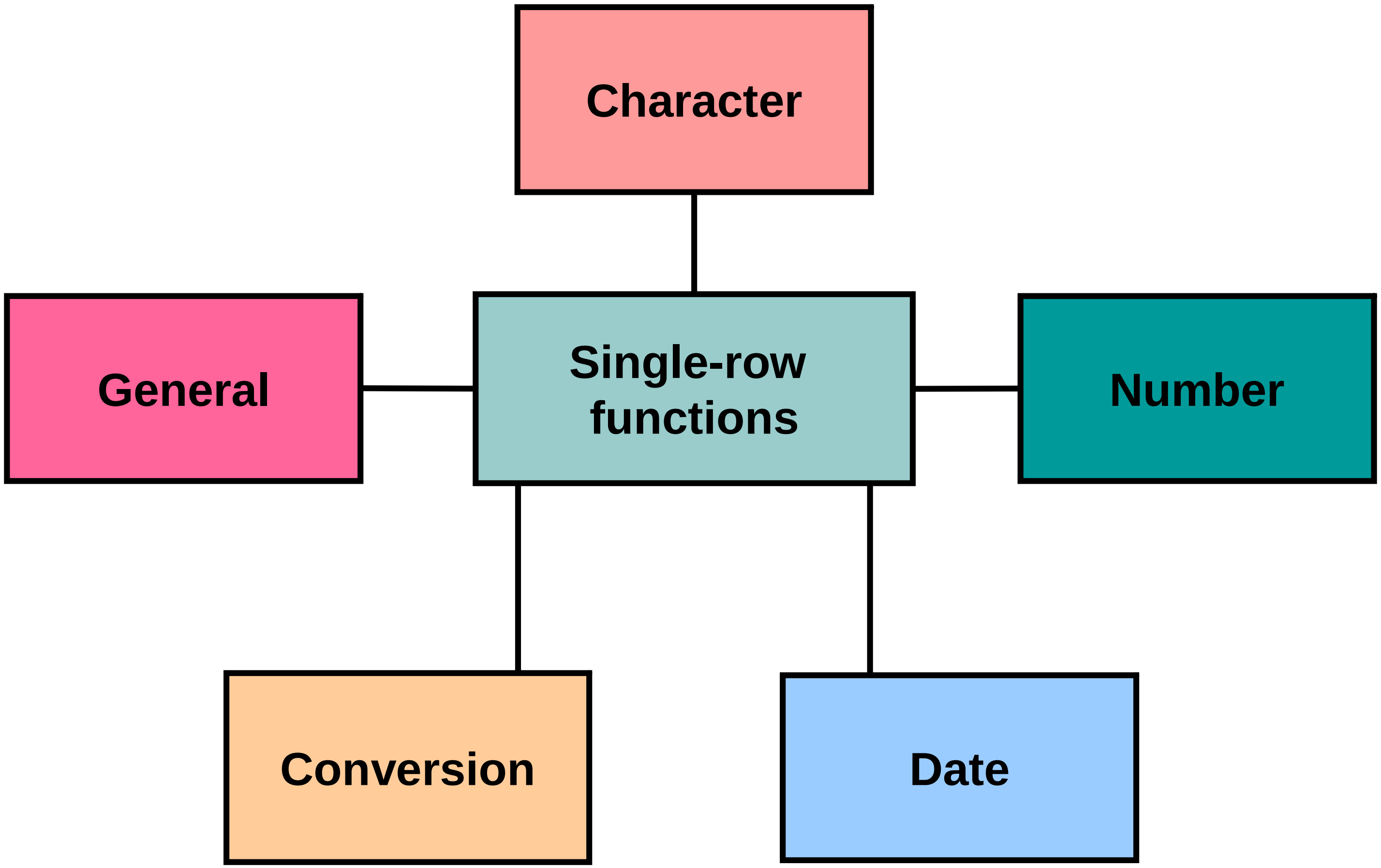
Single-Row Functions

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

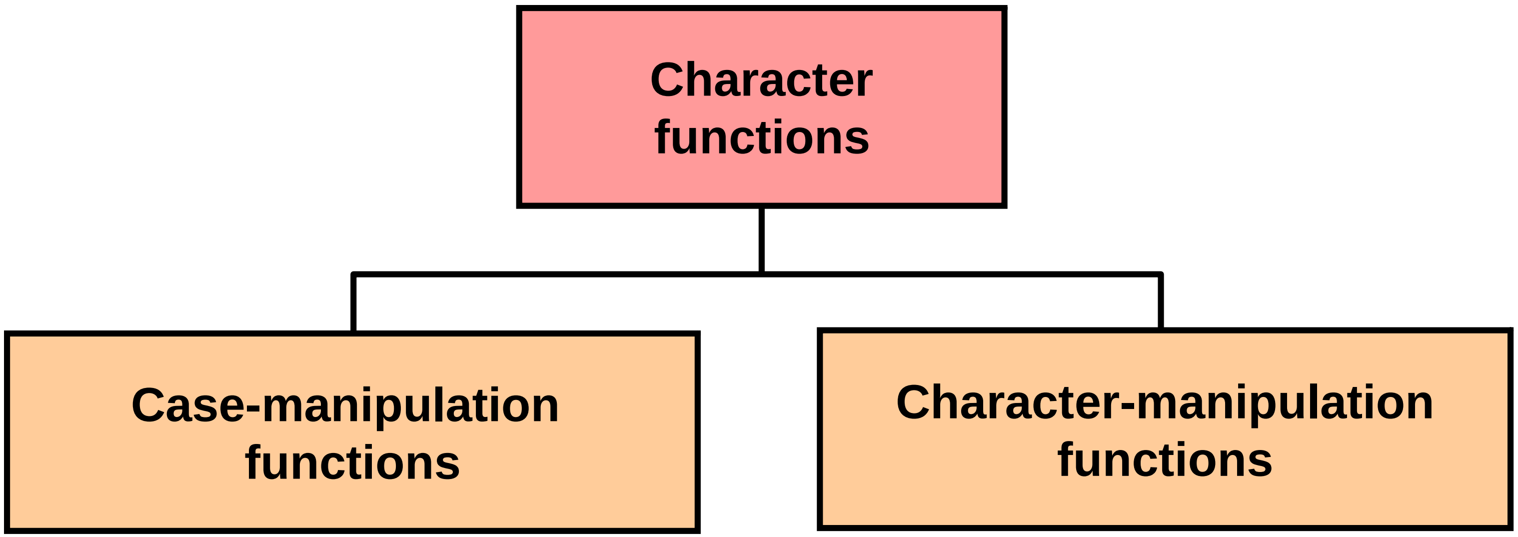
```
function_name [(arg1, arg2,...)]
```

Single-Row Functions



ORACLE

Character Functions



Case-manipulation functions:

`LOWER`
`UPPER`
`INITCAP`

Character-manipulation functions:

`CONCAT`
`SUBSTR`
`LENGTH`
`INSTR`
`LPAD | RPAD`
`TRIM`
`REPLACE`

Case-Manipulation Functions

These functions convert case for character strings:

Function	Result
`LOWER('SQL Course')`	`sql course`
`UPPER('SQL Course')`	`SQL COURSE`
`INITCAP('SQL Course')`	`Sql Course`

Using Case-Manipulation 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';
no rows selected
```

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

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
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'?
174	EllenAbel	SA_REP	4	0
176	JonathonTaylor	SA_REP	6	2
178	KimberelyGrant	SA_REP	5	3
202	PatFay	MK_REP	3	2

Number Functions

- `ROUND`: **Rounds value to specified decimal**
- `TRUNC`: **Truncates value to 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

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

ROUND(45.923,2)	ROUND(45.923,0)	ROUND(45.923,-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

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

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

Using the MOD Function

For all employees with 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)
Abel	11000	1000
Taylor	8600	3600
Grant	7000	2000

ORACLE

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
King	17-JUN-87
Whalen	17-SEP-87

ORACLE

Working with Dates

`SYSDATE` **is a function that returns:**

- **Date**
- **Time**

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
King	744.245395
Kochhar	626.102538
De Haan	453.245395

Date 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 ('11-JAN-94',6)`	`'11-JUL-94'`
`NEXT_DAY ('01-SEP-95','FRIDAY')`	`'08-SEP-95'`
`LAST_DAY ('01-FEB-95')`	`'28-FEB-95'`

ORACLE

Using Date Functions

Assume `SYSDATE = '25-JUL-03'`:

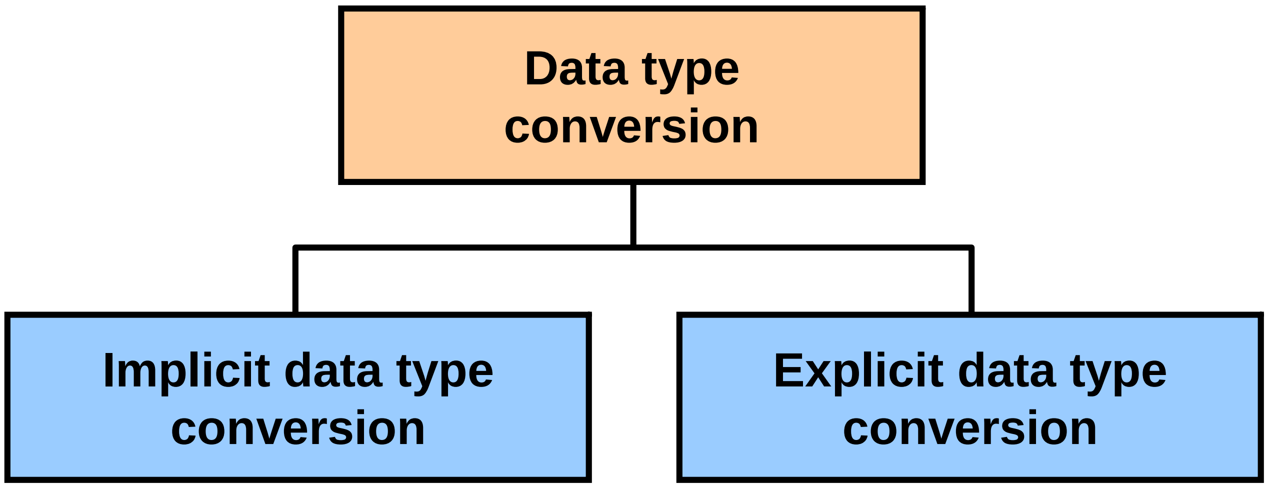
Function	Result
`ROUND(SYSDATE,'MONTH')`	`01-AUG-03`
`ROUND(SYSDATE ,'YEAR')`	`01-JAN-04`
`TRUNC(SYSDATE ,'MONTH')`	`01-JUL-03`
`TRUNC(SYSDATE ,'YEAR')`	`01-JAN-03`

Practice 3: Overview of Part 1

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

Conversion Functions



Implicit Data Type Conversion

For assignments, the Oracle server can automatically convert the following:

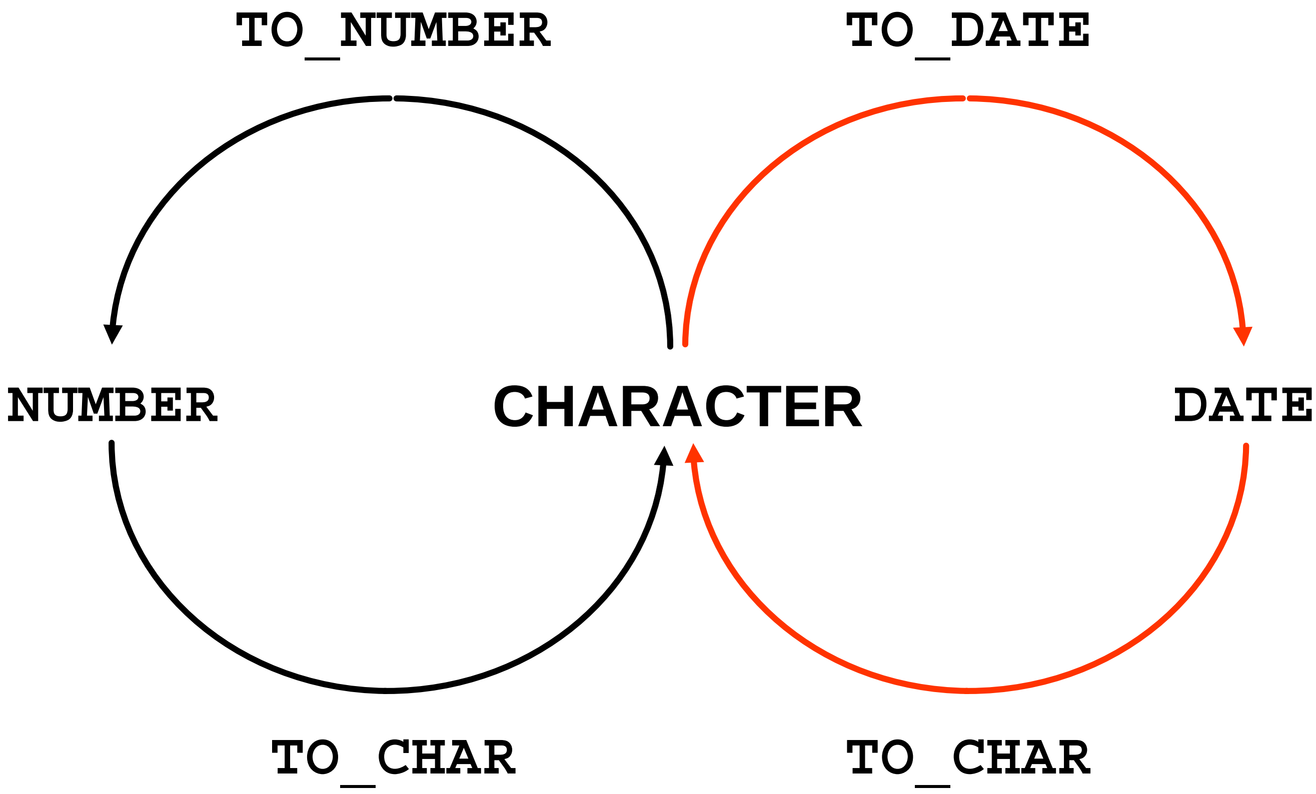
From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE
NUMBER	VARCHAR2
DATE	VARCHAR2

Implicit Data Type Conversion

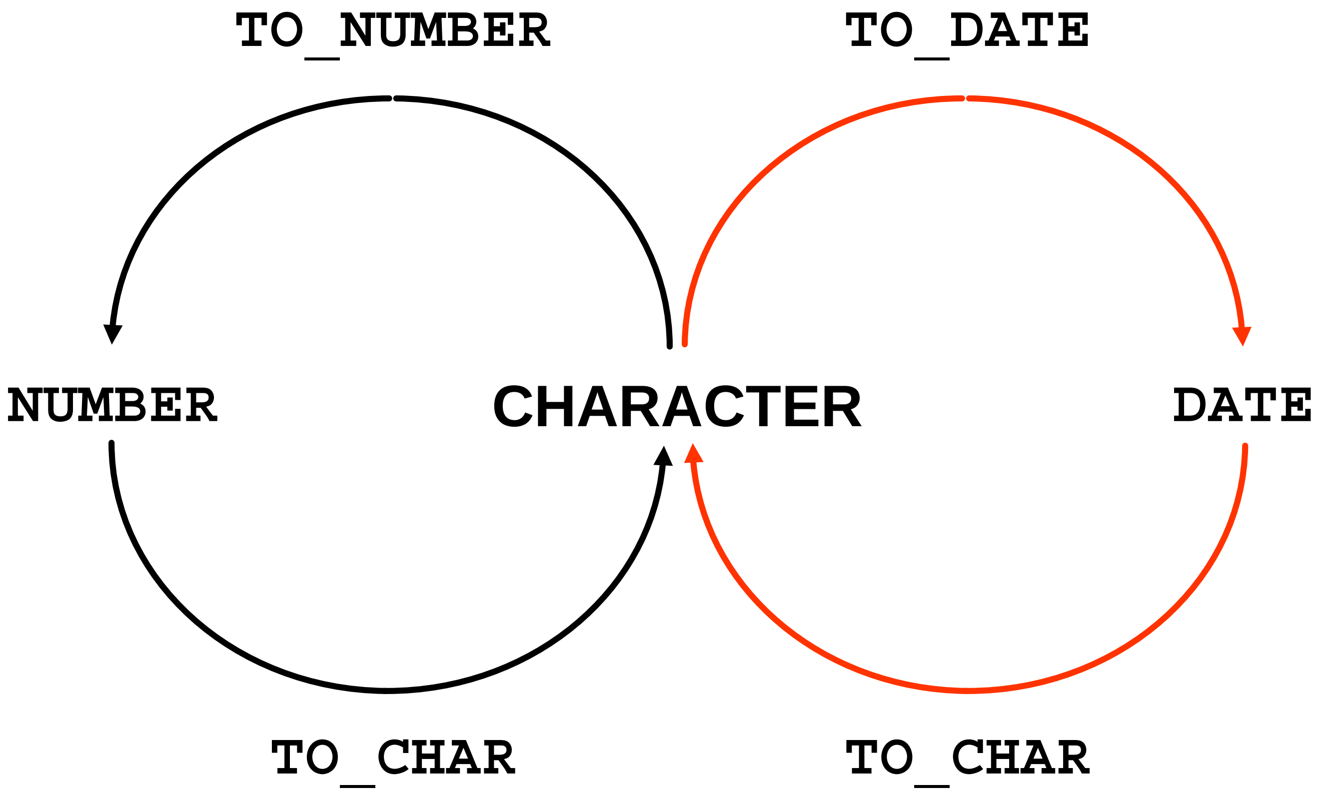
For expression evaluation, the Oracle Server can automatically convert the following:

From	To
`VARCHAR2 or CHAR`	`NUMBER`
`VARCHAR2 or CHAR`	`DATE`

Explicit Data Type Conversion



Explicit Data Type Conversion



Using the TO_CHAR Function with Dates

```
TO_CHAR(date, 'format_model')
```

The format model:

- Must be enclosed by 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

ORACLE

Elements of the Date Format Model

Element	Result
`YYYY`	Full year in numbers
`YEAR`	Year spelled out (in English)
`MM`	Two-digit value for 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 in 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
King	17 June 1987
Kochhar	21 September 1989
De Haan	13 January 1993
Hunold	3 January 1990
Ernst	21 May 1991
Lorentz	7 February 1999
Mourgos	16 November 1999

...

20 rows selected.

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 thousands indicator

ORACLE

Using the `TO_CHAR` Function with Numbers

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

SALARY
$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 matching for the character argument and date format model of a TO_DATE function.

`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

Example of RR Date Format

To find employees hired prior to 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(HIR
King	17-Jun-1987
Kochhar	21-Sep-1989
Whalen	17-Sep-1987

Nesting Functions

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

```
F3(F2(F1(col,arg1),arg2),arg3)
```

Step 1 = Result 1

Step 2 = Result 2

Step 3 = Result 3

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
Hunold	HUNOLD_US
Ernst	ERNST_US
Lorentz	LORENTZ_US

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;
```

1

2

LAST_NAME	SALARY	NVL(COMMISSION_PCT,0)	AN_SAL
King	24000	0	288000
Kochhar	17000	0	204000
De Haan	17000	0	204000
Hunold	9000	0	108000
Ernst	6000	0	72000
Lorentz	4200	0	50400
Mourgos	5800	0	69600
Rajs	3500	0	42000

...

20 rows selected.

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);
```

1

2

LAST_NAME	SALARY	COMMISSION_PCT	INCOME
Zlotkey	10500	.2	SAL+COMM
Abel	11000	.3	SAL+COMM
Taylor	8600	.2	SAL+COMM
Mourgos	5800		SAL
Rajs	3500		SAL
Davies	3100		SAL
Matos	2600		SAL
Vargas	2500		SAL

8 rows selected.

1

2

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
Steven	6	King	4	6
Neena	5	Kochhar	7	5
Lex	3	De Haan	7	3
Alexander	9	Hunold	6	9
Bruce	5	Ernst	5	
Diana	5	Lorentz	7	5
Kevin	5	Mourgos	7	5
Trenna	6	Rajs	4	6
Curtis	6	Davies	6	

...

20 rows selected.

1 2 3

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,
       COALESCE(manager_id,commission_pct, -1) comm
FROM   employees
ORDER BY commission_pct;
```

LAST_NAME	COMM
Grant	149
Zlotkey	100
Taylor	149
Abel	149
King	-1
Kochhar	100
De Haan	100

...

20 rows selected.

Conditional Expressions

- Provide the use of 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
...			
Lorentz	IT_PROG	4200	4620
Mourgos	ST_MAN	5800	5800
Rajs	ST_CLERK	3500	4025
...			
Gietz	AC_ACCOUNT	8300	8300

20 rows selected.

`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
...			
Lorentz	IT_PROG	4200	4620
Mourgos	ST_MAN	5800	5800
Rajs	ST_CLERK	3500	4025
...			
Gietz	AC_ACCOUNT	8300	8300

20 rows selected.

ORACLE

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;
```

Summary

In this lesson, you should have learned how to:

- Perform calculations on data using functions
- Modify individual data items using functions
- Manipulate output for groups of rows using functions
- Alter date formats for display using functions
- Convert column data types using functions
- Use `NVL` functions
- Use IF-THEN-ELSE logic

Practice 3: Overview of Part 2

This practice covers the following topics:

- Creating queries that require the use of numeric, character, and date functions
- Using concatenation with functions
- Writing case-insensitive queries to test the usefulness of character functions
- Performing calculations of years and months of service for an employee
- Determining the review date for an employee

4

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

What Are Group Functions?

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

EMPLOYEES

DEPARTMENT_ID	SALARY
90	24000
90	17000
90	17000
60	9000
60	6000
60	4200
50	5800
50	3500
50	3100
50	2600
50	2500
80	10500
80	11000
80	8600
	7000
10	4400

...

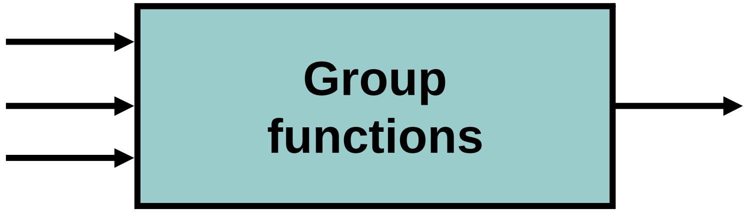
20 rows selected.

Maximum salary in EMPLOYEES table

MAX(SALARY)
24000

Types of Group Functions

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



Group Functions: Syntax

```
SELECT      [column,] group_function(column), ...
FROM        table
[WHERE      condition]
[GROUP BY   column]
[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)
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_	MAX(HIRE_
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(*)
5

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

2

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

COUNT(COMMISSION_PCT)
3

ORACLE

Using the `DISTINCT` Keyword

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

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

COUNT(DISTINCTDEPARTMENT_ID)
7

Group Functions and Null Values

Group functions ignore null values in the column:

1

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

AVG(COMMISSION_PCT)
.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))
.0425

Creating Groups of Data

EMPLOYEES

DEPARTMENT_ID	SALARY
10	4400
20	13000
20	6000
50	5800
50	3500
50	3100
50	2500
50	2600
60	9000
60	6000
60	4200
80	10500
80	8600
80	11000
90	24000
90	17000

...

20 rows selected.

4400

9500

3500

6400

10033

Average salary in EMPLOYEES table for each department

DEPARTMENT_ID	AVG(SALARY)
10	4400
20	9500
50	3500
60	6400
80	10033.3333
90	19333.3333
110	10150
	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)
10	4400
20	9500
50	3500
60	6400
80	10033.3333
90	19333.3333
110	10150
	7000

8 rows selected.

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)
4400
9500
3500
6400
10033.3333
19333.3333
10150
7000

Grouping by More Than One Column

EMPLOYEES

DEPARTMENT_ID	JOB_ID	SALARY
90	AD_PRES	24000
90	AD_VP	17000
90	AD_VP	17000
60	IT_PROG	9000
60	IT_PROG	6000
60	IT_PROG	4200
50	ST_MAN	5800
50	ST_CLERK	3500
50	ST_CLERK	3100
50	ST_CLERK	2600
50	ST_CLERK	2500
80	SA_MAN	10500
80	SA_REP	11000
80	SA_REP	8600

...

20	MK_REP	6000
110	AC_MGR	12000
110	AC_ACCOUNT	8300

20 rows selected.

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

DEPARTMENT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
20	MK_MAN	13000
20	MK_REP	6000
50	ST_CLERK	11700
50	ST_MAN	5800
60	IT_PROG	19200
80	SA_MAN	10500
80	SA_REP	19600
90	AD_PRES	24000
90	AD_VP	34000
110	AC_ACCOUNT	8300
110	AC_MGR	12000
	SA_REP	7000

13 rows selected.

Using the GROUP BY Clause on Multiple Columns

```
SELECT    department_id dept_id, job_id, SUM(salary)
FROM      employees
GROUP BY department_id, job_id ;
```

DEPT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
20	MK_MAN	13000
20	MK_REP	6000
50	ST_CLERK	11700
50	ST_MAN	5800
60	IT_PROG	19200
80	SA_MAN	10500
80	SA_REP	19600
90	AD_PRES	24000
90	AD_VP	34000
110	AC_ACCOUNT	8300
110	AC_MGR	12000
	SA_REP	7000

13 rows selected.

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;
```

```
SELECT department_id, COUNT(last_name)
       *
ERROR at line 1:
ORA-00937: not a single-group group function
```

Column missing in the `GROUP BY` clause

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;
```

```
WHERE  AVG(salary) > 8000
       *
ERROR at line 3:
ORA-00934: group function is not allowed here
```

Cannot use the `WHERE` clause to restrict groups

Restricting Group Results

EMPLOYEES

DEPARTMENT_ID	SALARY
90	24000
90	17000
90	17000
60	9000
60	6000
60	4200
50	5800
50	3500
50	3100
50	2600
50	2500
80	10500
80	11000
80	8600
…	
20	6000
110	12000
110	8300

20 rows selected.

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

DEPARTMENT_ID	MAX(SALARY)
20	13000
80	11000
90	24000
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];
```

ORACLE

Using the `HAVING` Clause

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

DEPARTMENT_ID	MAX(SALARY)
20	13000
80	11000
90	24000
110	12000

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
IT_PROG	19200
AD_PRES	24000
AD_VP	34000

Nesting Group Functions

Display the maximum average salary:

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

MAX(AVG(SALARY))
19333.3333

Summary

In this lesson, you should have learned how to:

- Use the group functions `COUNT`, `MAX`, `MIN`, 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 4: 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**

5

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 non-equijoins
- 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

ORACLE

Obtaining Data from Multiple Tables

EMPLOYEES

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
...		
202	Fay	20
205	Higgins	110
206	Gietz	110

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
60	IT	1400
80	Sales	2500
90	Executive	1700
110	Accounting	1700
190	Contracting	1700

EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
200	10	Administration
201	20	Marketing
202	20	Marketing
...		
102	90	Executive
205	110	Accounting
206	110	Accounting

ORACLE

Types of Joins

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

- **Cross joins**
- **Natural joins**
- **`USING` clause**
- **Full (or two-sided) outer joins**
- **Arbitrary join conditions for outer 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];
```

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
60	IT	1400	Southlake
50	Shipping	1500	South San Francisco
10	Administration	1700	Seattle
90	Executive	1700	Seattle
110	Accounting	1700	Seattle
190	Contracting	1700	Seattle
20	Marketing	1800	Toronto
80	Sales	2500	Oxford

8 rows selected.

Creating Joins with the `USING` Clause

- If several columns have the same names but the data types do not match, the `NATURAL JOIN` clause can be modified with the `USING` clause to specify the columns that should be used for an equijoin.
- Use the `USING` clause to match only one column when more than one column matches.
- Do not use a table name or alias in the referenced columns.
- The `NATURAL JOIN` and `USING` clauses are mutually exclusive.

Joining Column Names

EMPLOYEES

EMPLOYEE_ID	DEPARTMENT_ID
200	10
201	20
202	20
124	50
141	50
142	50
143	50
144	50
103	60
104	60
107	60
149	80
174	80
176	80

…

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
20	Marketing
50	Shipping
50	Shipping
50	Shipping
50	Shipping
50	Shipping
60	IT
60	IT
60	IT
80	Sales
80	Sales
80	Sales

…

Foreign key **Primary key**

Retrieving Records with the `USING` Clause

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

EMPLOYEE_ID	LAST_NAME	LOCATION_ID	DEPARTMENT_ID
200	Whalen	1700	10
201	Hartstein	1800	20
202	Fay	1800	20
124	Mourgos	1500	50
141	Rajs	1500	50
142	Davies	1500	50
144	Vargas	1500	50
143	Matos	1500	50

...

19 rows selected.

Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Use table prefixes to improve performance.
- Use column aliases to distinguish columns that have identical names but reside in different tables.
- Do not use aliases on columns that are identified in the `USING` clause and listed elsewhere in the SQL statement.

Using Table Aliases

- Use table aliases to simplify queries.
- Use table aliases to improve performance.

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

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	LOCATION_ID
200	Whalen	10	10	1700
201	Hartstein	20	20	1800
202	Fay	20	20	1800
124	Mourgos	50	50	1500
141	Rajs	50	50	1500
142	Davies	50	50	1500
143	Matos	50	50	1500

...

19 rows selected.

Self-Joins Using the ON Clause

EMPLOYEES (WORKER)

EMPLOYEE_ID	LAST_NAME	MANAGER_ID
100	King	
101	Kochhar	100
102	De Haan	100
103	Hunold	102
104	Ernst	103
107	Lorentz	103
124	Mourgos	100

...

EMPLOYEES (MANAGER)

EMPLOYEE_ID	LAST_NAME
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst
107	Lorentz
124	Mourgos

...

MANAGER_ID in the WORKER table is equal to EMPLOYEE_ID in the MANAGER table.

Self-Joins Using the ON Clause

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

EMP	MGR
Hartstein	King
Zlotkey	King
Mourgos	King
De Haan	King
Kochhar	King

...

19 rows selected.

ORACLE

Applying Additional Conditions to a Join

```
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 ;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
174	Abel	80	80	2500
176	Taylor	80	80	2500

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
103	Southlake	IT
104	Southlake	IT
107	Southlake	IT
124	South San Francisco	Shipping
141	South San Francisco	Shipping
142	South San Francisco	Shipping
143	South San Francisco	Shipping
144	South San Francisco	Shipping

...

19 rows selected.

Non-Equijoins

EMPLOYEES

LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Hunold	9000
Ernst	6000
Lorentz	4200
Mourgos	5800
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500
Zlotkey	10500
Abel	11000
Taylor	8600

…

20 rows selected.

JOB_GRADES

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

Salary in the `EMPLOYEES` table must be between lowest salary and highest salary in the `JOB_GRADES` table.

Retrieving Records with Non-Equijoins

```
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	GRA
Matos	2600	A
Vargas	2500	A
Lorentz	4200	B
Mourgos	5800	B
Rajs	3500	B
Davies	3100	B
Whalen	4400	B
Hunold	9000	C
Ernst	6000	C

...

20 rows selected.

Outer Joins

DEPARTMENTS

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

8 rows selected.

EMPLOYEES

DEPARTMENT_ID	LAST_NAME
90	King
90	Kochhar
90	De Haan
60	Hunold
60	Ernst
60	Lorentz
50	Mourgos
50	Rajs
50	Davies
50	Matos
50	Vargas
80	Zlotkey

...

20 rows selected.

There are no employees in department 190.

`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) tables 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
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing
...		
De Haan	90	Executive
Kochhar	90	Executive
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		

20 rows selected.

RIGHT OUTER JOIN

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

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing
Davies	50	Shipping
...		
Kochhar	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
	190	Contracting

20 rows selected.

ORACLE

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) ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing
...		
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		
	190	Contracting

21 rows selected.

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
100	King	90
101	Kochhar	90

...

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
202	Fay	20
205	Higgins	110
206	Gietz	110

20 rows selected.

DEPARTMENTS (8 rows)

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
60	IT	1400
80	Sales	2500
90	Executive	1700
110	Accounting	1700
190	Contracting	1700

8 rows selected.

Cartesian product:
20 x 8 = 160 rows

EMPLOYEE_ID	DEPARTMENT_ID	LOCATION_ID
100	90	1700
101	90	1700
102	90	1700
103	60	1700
104	60	1700
107	60	1700

...

160 rows selected.

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
King	Administration
Kochhar	Administration
De Haan	Administration
Hunold	Administration

...

160 rows selected.

ORACLE

Summary

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

- Equijoins
- Non-equijoins
- Outer joins
- Self-joins
- Cross joins
- Natural joins
- Full (or two-sided) outer joins

Practice 5: 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 subqueries can solve
- List the types of subqueries
- Write single-row and multiple-row subqueries

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

Using a Subquery

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

LAST_NAME
King
Kochhar
De Haan
Hartstein
Higgins

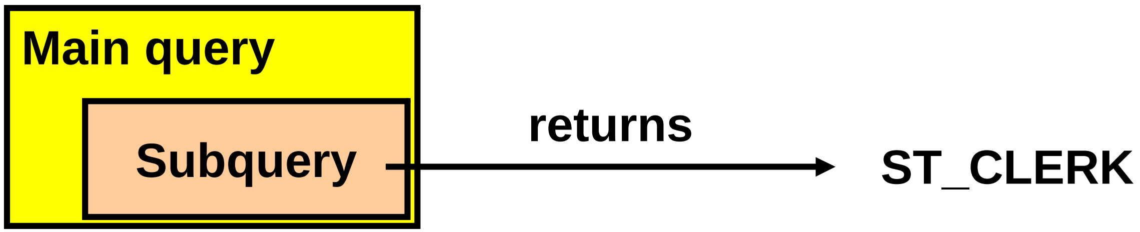
ORACLE

Guidelines for Using Subqueries

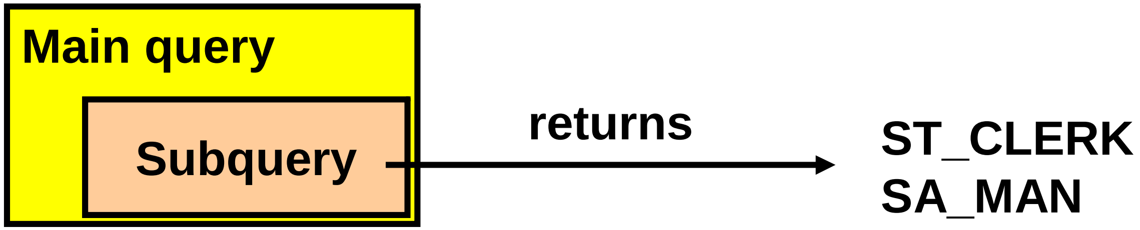
- Enclose subqueries in parentheses.
- Place subqueries on the right side of the comparison condition.
- The `ORDER BY` clause in the subquery is not needed unless you are performing Top-N analysis.
- Use single-row operators with single-row subqueries, and use multiple-row operators with multiple-row subqueries.

Types of Subqueries

- **Single-row subquery**



- **Multiple-row 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

ORACLE

Executing Single-Row Subqueries

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  job_id =                               ST_CLERK
                (SELECT job_id
                 FROM   employees
                 WHERE  employee_id = 141)
AND    salary >                                  2600
                (SELECT salary
                 FROM   employees
                 WHERE  employee_id = 143);
```

LAST_NAME	JOB_ID	SALARY
Rajs	ST_CLERK	3500
Davies	ST_CLERK	3100

Using Group Functions in a Subquery

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

2500

LAST_NAME	JOB_ID	SALARY
Vargas	ST_CLERK	2500

ORACLE

The `HAVING` Clause with Subqueries

- The Oracle server executes 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

What Is Wrong with This Statement?

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

```
ERROR at line 4:
ORA-01427: single-row subquery returns more than
one row
```

Single-row operator with multiple-row subquery

Will This Statement Return Rows?

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

```
no rows selected
```

Subquery returns no values.

Multiple-Row Subqueries

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

Operator	Meaning
`IN`	Equal to any member in the list
`ANY`	Compare value to each value returned by the subquery
`ALL`	Compare value to every value returned by the subquery

Using the ANY Operator in Multiple-Row Subqueries

```
SELECT employee_id, last_name, job_id, salary
FROM   employees        9000, 6000, 4200
WHERE  salary < ANY
                    (SELECT salary
                     FROM   employees
                     WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
124	Mourgos	ST_MAN	5800
141	Rajs	ST_CLERK	3500
142	Davies	ST_CLERK	3100
143	Matos	ST_CLERK	2600
144	Vargas	ST_CLERK	2500

...

10 rows selected.

Using the ALL Operator in Multiple-Row Subqueries

```
SELECT employee_id, last_name, job_id, salary
FROM   employees          9000, 6000, 4200
WHERE  salary < ALL
                    (SELECT salary
                     FROM   employees
                     WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';
```

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

ORACLE

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);

no rows selected
```

Summary

In this lesson, you should have learned how to:

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

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

Practice 6: Overview

This practice covers the following topics:

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

7

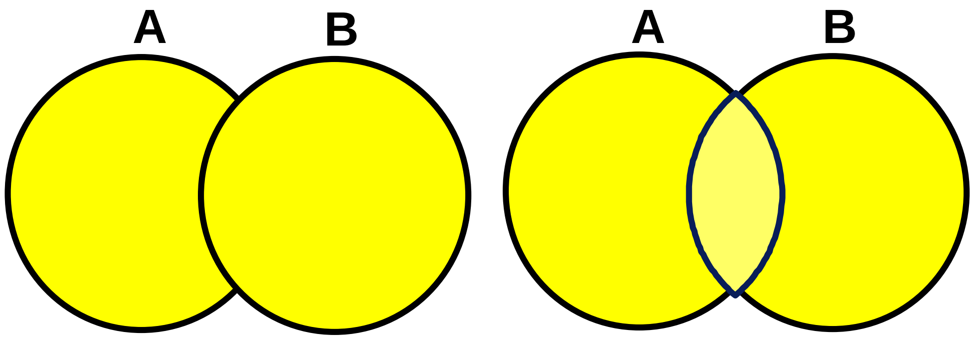
Using the Set Operators

Objectives

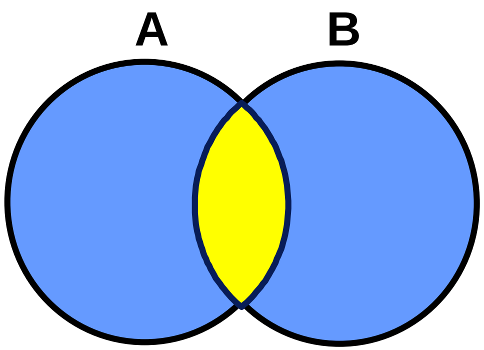
After completing this lesson, you should be able to do the following:

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

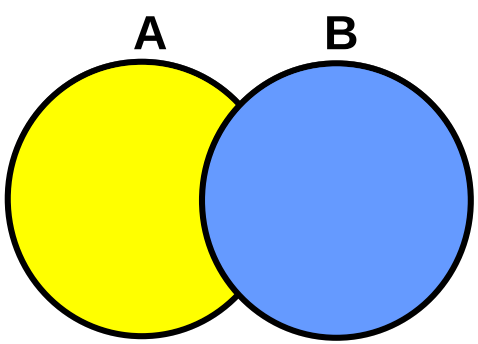
Set Operators



`UNION/UNION ALL`



`INTERSECT`



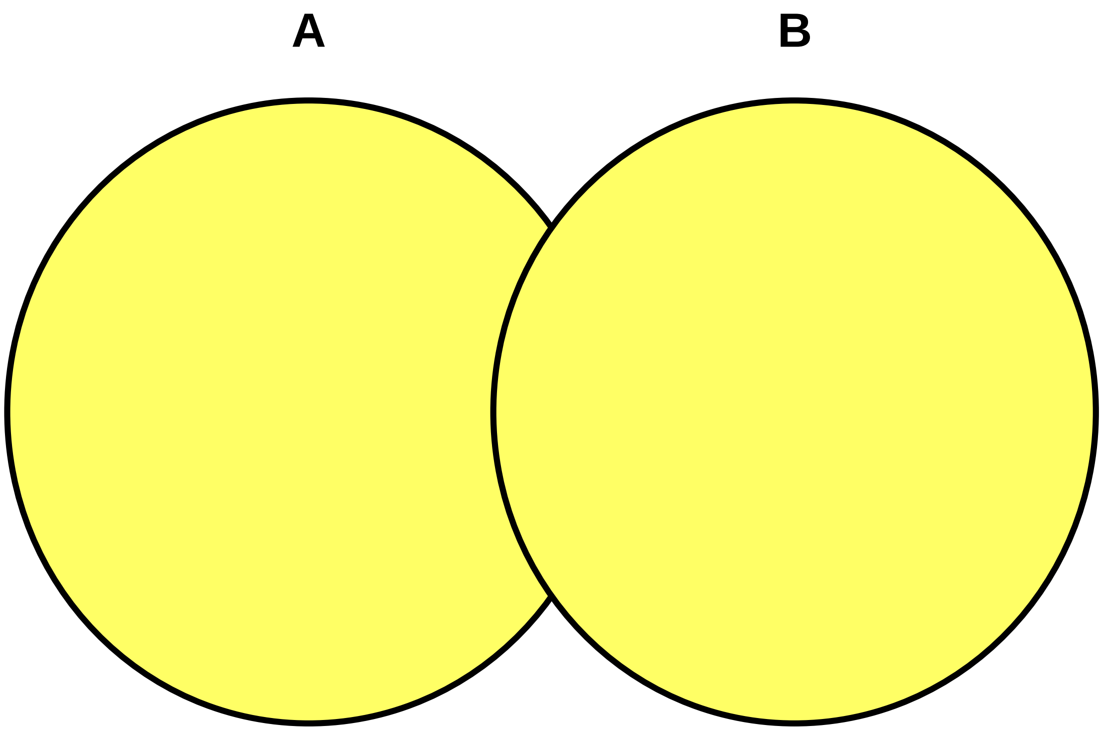
`MINUS`

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**

`UNION` Operator



The `UNION` operator returns results from both queries after eliminating duplications.

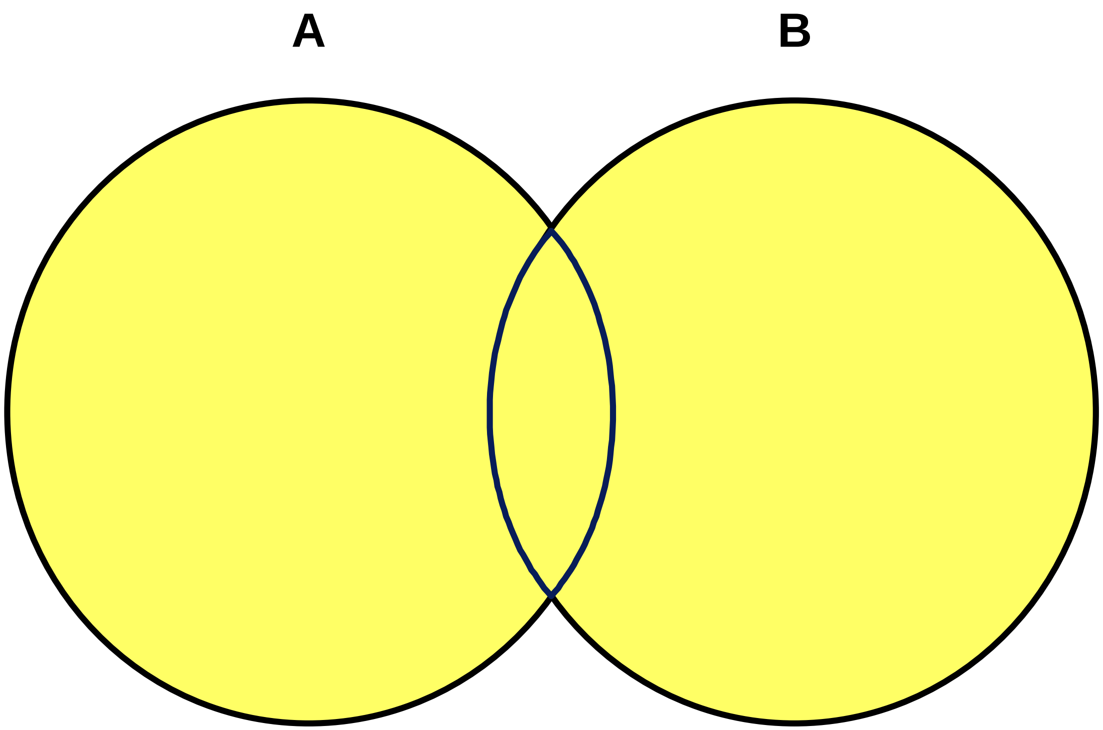
Using the UNION Operator

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
100	AD_PRES
101	AC_ACCOUNT
...	
200	AC_ACCOUNT
200	AD_ASST
...	
205	AC_MGR
206	AC_ACCOUNT

`UNION ALL` Operator



The `UNION ALL` operator returns results 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
100	AD_PRES	90
101	AD_VP	90

...

200	AD_ASST	10
200	AD_ASST	90
200	AC_ACCOUNT	90

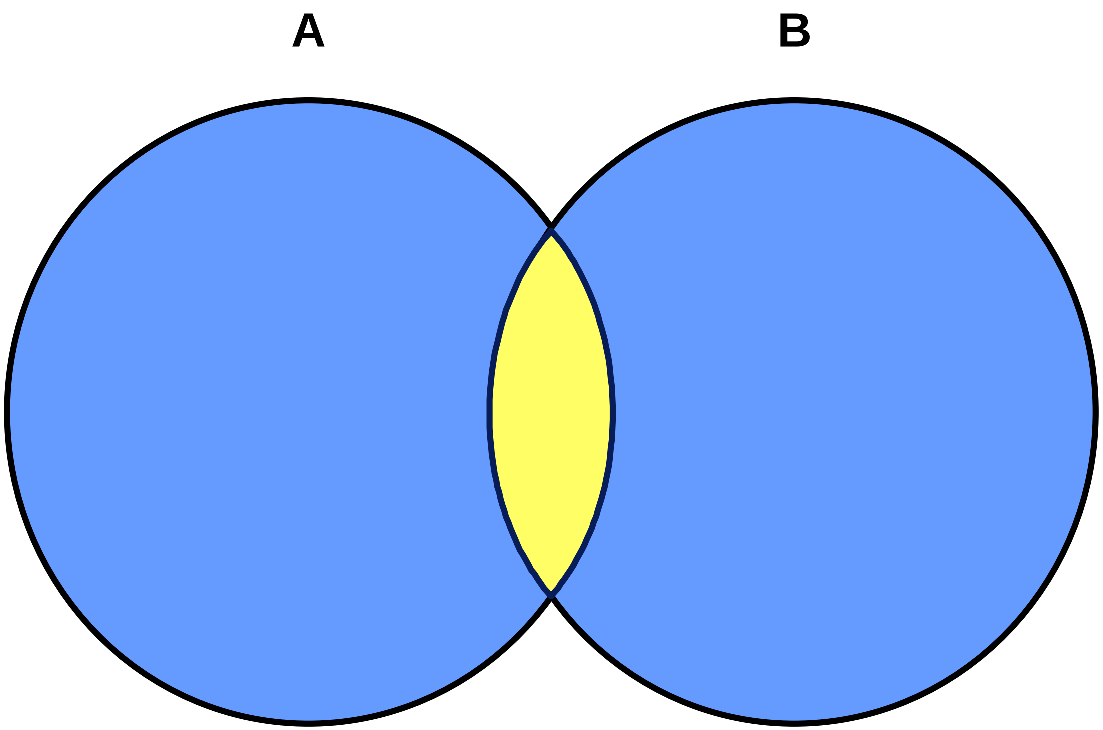
...

205	AC_MGR	110
206	AC_ACCOUNT	110

30 rows selected.

ORACLE

`INTERSECT` Operator



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

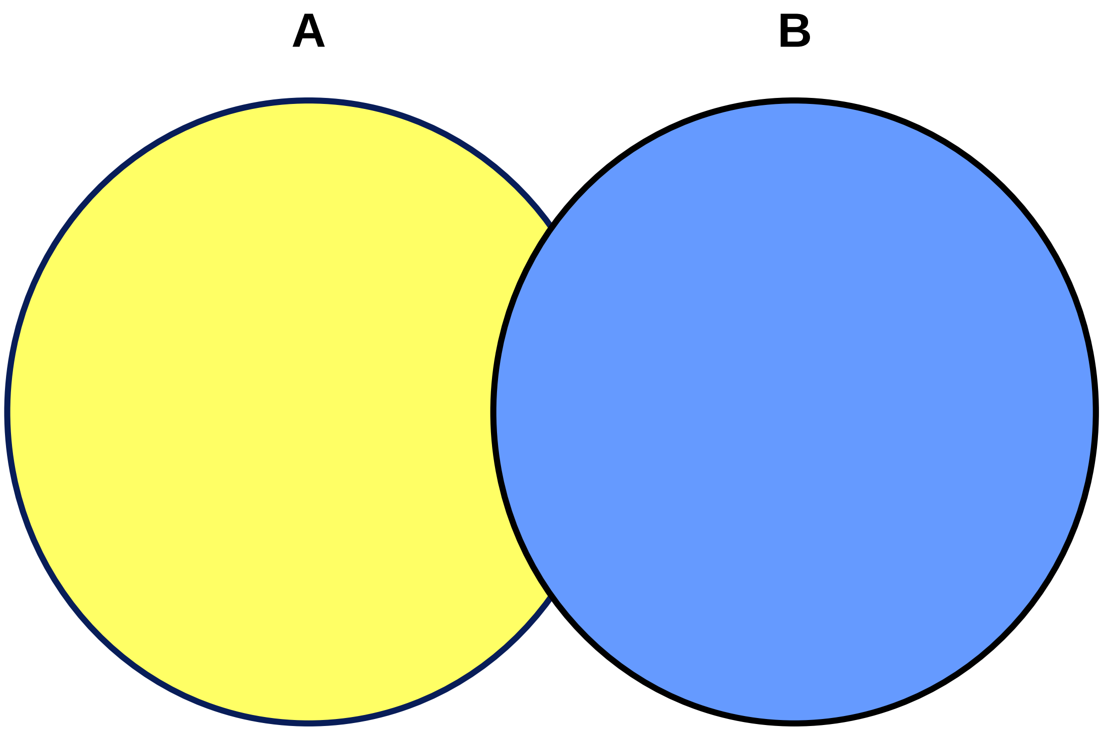
Using the INTERSECT Operator

Display the employee IDs and job IDs of those employees who currently have a job title that is the same as their job title when they were initially hired (that is, they changed jobs but have now gone back to doing their original job).

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

EMPLOYEE_ID	JOB_ID
176	SA_REP
200	AD_ASST

`MINUS` Operator



The `MINUS` operator returns rows in the first query that are not present in the second query.

MINUS Operator

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

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

EMPLOYEE_ID	JOB_ID
100	AD_PRES
101	AD_VP
102	AD_VP
103	IT_PROG
...	
201	MK_MAN
202	MK_REP
205	AC_MGR
206	AC_ACCOUNT

18 rows selected.

ORACLE

Set Operator Guidelines

- The expressions in the `SELECT` lists must match in number and data type.
- Parentheses can be used to alter the sequence of execution.
- The `ORDER BY` clause:
  - Can appear only at the very end of the statement
  - Will accept the column name, aliases from the first `SELECT` statement, or the positional notation

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`.

Matching the SELECT Statements

Using the UNION operator, display the department ID, location, and hire date for all employees.

```
SELECT department_id, TO_NUMBER(null)
       location, hire_date
FROM   employees
UNION
SELECT department_id, location_id,  TO_DATE(null)
FROM   departments;
```

DEPARTMENT_ID	LOCATION	HIRE_DATE
10	1700	
10		17-SEP-87
20	1800	
20		17-FEB-96
...		
110	1700	
110		07-JUN-94
190	1700	
		24-MAY-99

27 rows selected.

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
100	AD_PRES	24000
101	AC_ACCOUNT	0
101	AC_MGR	0
...		
205	AC_MGR	12000
206	AC_ACCOUNT	8300

30 rows selected.

ORACLE

Controlling the Order of Rows

Produce an English sentence using two `UNION` operators.

```
COLUMN a_dummy NOPRINT
SELECT 'sing' AS "My dream", 3 a_dummy
FROM dual
UNION
SELECT 'I''d like to teach', 1 a_dummy
FROM dual
UNION
SELECT 'the world to', 2 a_dummy
FROM dual
ORDER BY a_dummy;
```

My dream
I'd like to teach
the world to
sing

Summary

In this lesson, you should have learned how to:

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

Practice 7: Overview

In this practice, you use the set operators to create reports:

- **Using the `UNION` operator**
- **Using the `INTERSECTION` operator**
- **Using the `MINUS` operator**

8

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

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

70	Public Relations	100	1700

New row

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		1700

Insert new row into the `DEPARTMENTS` table

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		1700
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 row created.
```

- Enclose character and date values in single quotation marks.

Inserting Rows with Null Values

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

```
INSERT INTO   departments (department_id,
                           department_name )
VALUES        (30, 'Purchasing');
1 row created.
```

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

```
INSERT INTO   departments
VALUES        (100, 'Finance', NULL, NULL);
1 row created.
```

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, 100);
1 row created.
```

Inserting Specific Date Values

- Add a new employee.

```
INSERT INTO employees
VALUES      (114,
             'Den', 'Raphealy',
             'DRAPHEAL', '515.127.4561',
             TO_DATE('FEB 3, 1999', 'MON DD, YYYY'),
             'AC_ACCOUNT', 11000, NULL, 100, 30);
1 row created.
```

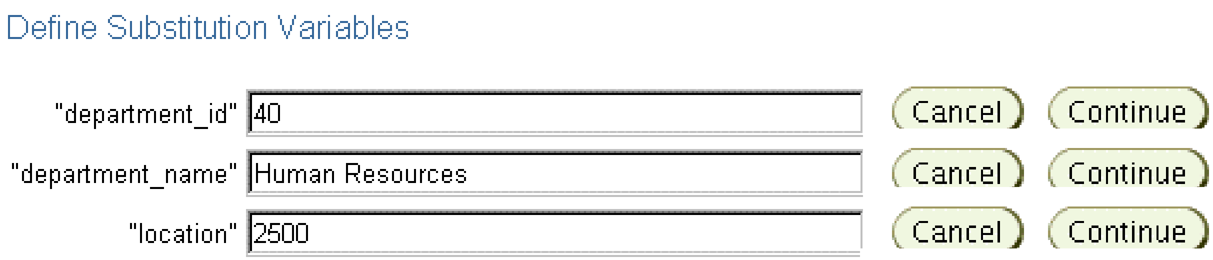
- Verify your addition.

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_P
114	Den	Raphealy	DRAPHEAL	515.127.4561	03-FEB-99	AC_ACCOUNT	11000	

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);
```




```
1 row created.
```

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 created.
```

- Do not use the `VALUES` clause.
- Match the number of columns in the `INSERT` clause to those in the subquery.

Changing Data in a Table

EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID	COMMISSION_P
100	Steven	King	SKING	17-JUN-87	AD_PRES	24000	90	
101	Neena	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	17000	90	
102	Lex	De Haan	LDEHAAN	13-JAN-93	AD_VP	17000	90	
103	Alexander	Hunold	AHUNOLD	03-JAN-90	IT_PROG	9000	60	
104	Bruce	Ernst	BERNST	21-MAY-91	IT_PROG	6000	60	
107	Diana	Lorentz	DLORENTZ	07-FEB-99	IT_PROG	4200	60	
124	Kevin	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	5800	50	

Update rows in the `EMPLOYEES` table:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID	COMMISSIO
100	Steven	King	SKING	17-JUN-87	AD_PRES	24000	90	
101	Neena	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	17000	90	
102	Lex	De Haan	LDEHAAN	13-JAN-93	AD_VP	17000	90	
103	Alexander	Hunold	AHUNOLD	03-JAN-90	IT_PROG	9000	30	
104	Bruce	Ernst	BERNST	21-MAY-91	IT_PROG	6000	30	
107	Diana	Lorentz	DLORENTZ	07-FEB-99	IT_PROG	4200	30	
124	Kevin	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	5800	50	

`UPDATE` Statement Syntax

- Modify existing rows 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

- Specific row or rows are modified if you specify the `WHERE` clause:

```
UPDATE employees
SET    department_id = 70
WHERE  employee_id = 113;
1 row updated.
```

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

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

Updating Two Columns with a Subquery

Update employee 114's job and salary to match that 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    =  114;
1 row updated.
```

ORACLE

Updating Rows Based on Another Table

Use subqueries in `UPDATE` statements to update rows in a table based on values from another table:

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

Removing a Row from a Table

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing		
100	Finance		
50	Shipping	124	1500
60	IT	103	1400

Delete a row from the `DEPARTMENTS` table:

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing		
50	Shipping	124	1500
60	IT	103	1400

ORACLE

`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 row 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 subqueries in `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 row 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;
```

Using a Subquery in an `INSERT` Statement

```
INSERT INTO
        (SELECT employee_id, last_name,
                email, hire_date, job_id, salary,
                department_id
         FROM   employees
         WHERE  department_id = 50)
VALUES (99999, 'Taylor', 'DTAYLOR',
        TO_DATE('07-JUN-99', 'DD-MON-RR'),
        'ST_CLERK', 5000, 50);

1 row created.
```

Using a Subquery in an `INSERT` Statement

Verify the results:

```
SELECT employee_id, last_name, email, hire_date,
       job_id, salary, department_id
FROM   employees
WHERE  department_id = 50;
```

EMPLOYEE_ID	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID
124	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	5800	50
141	Rajs	TRAJS	17-OCT-95	ST_CLERK	3500	50
142	Davies	CDAVIES	29-JAN-97	ST_CLERK	3100	50
143	Matos	RMATOS	15-MAR-98	ST_CLERK	2600	50
144	Vargas	PVARGAS	09-JUL-98	ST_CLERK	2500	50
99999	Taylor	DTAYLOR	07-JUN-99	ST_CLERK	5000	50

6 rows selected.

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

- 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 *i*SQL*Plus.
  - The system crashes.

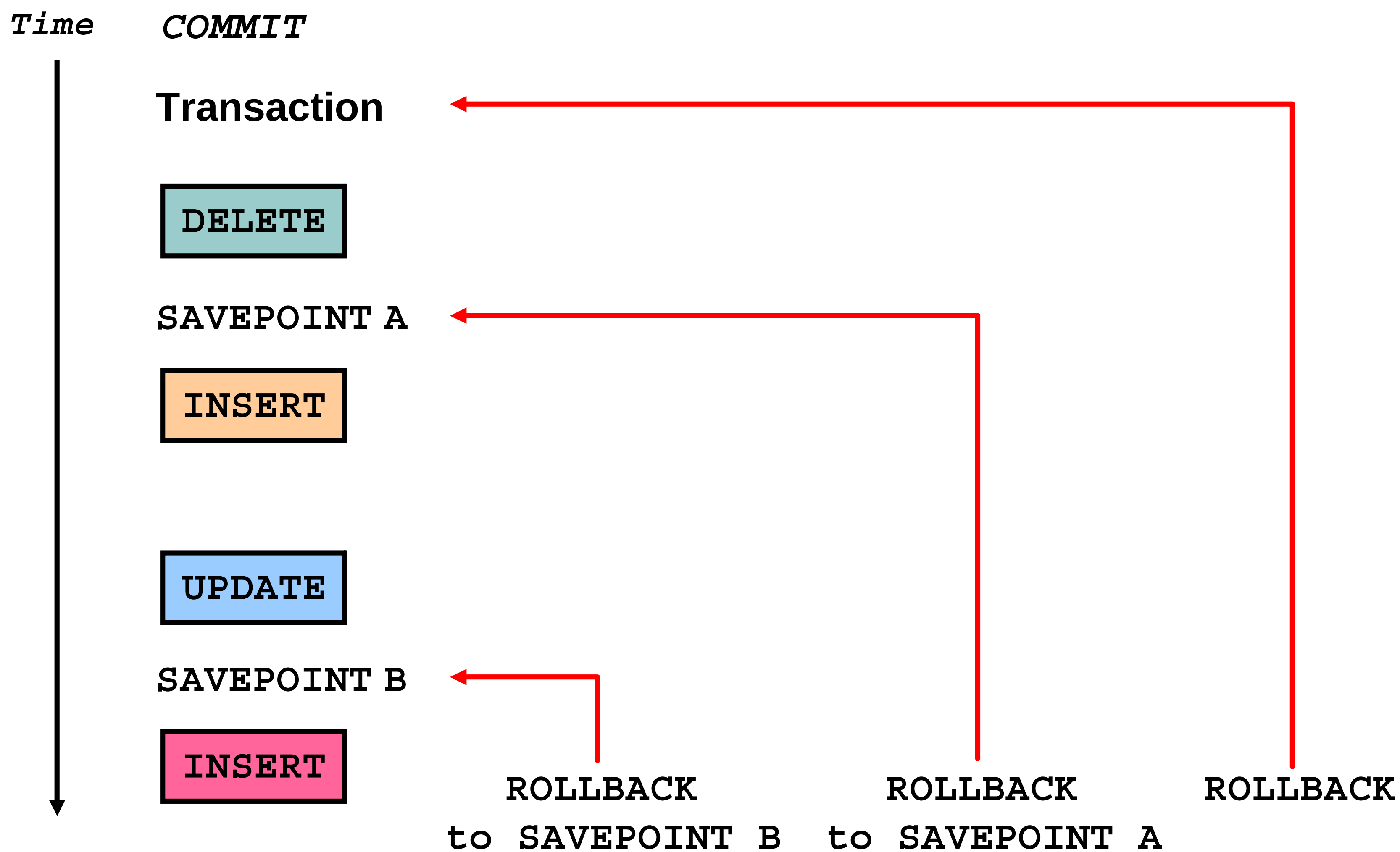
ORACLE

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

Controlling Transactions



Rolling Back Changes to a Marker

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

```
UPDATE...
SAVEPOINT update_done;
Savepoint created.
INSERT...
ROLLBACK TO update_done;
Rollback complete.
```

Implicit Transaction Processing

- An automatic commit occurs under the following circumstances:
  - DDL statement is issued
  - DCL statement is issued
  - Normal exit from *i*SQL*Plus, without explicitly issuing `COMMIT` or `ROLLBACK` statements
- An automatic rollback occurs under an abnormal termination of *i*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 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 made permanent in the database.
- The previous state of the data is permanently lost.
- 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 row deleted.

INSERT INTO departments
VALUES (290, 'Corporate Tax', NULL, 1700);
1 row created.
```

- **Commit the changes:**

```
COMMIT;
Commit complete.
```

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;
22 rows deleted.
ROLLBACK ;
Rollback complete.
```

State of the Data After ROLLBACK

```
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

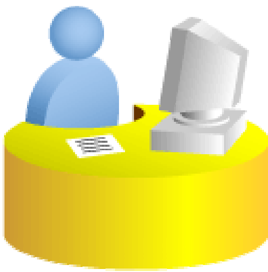
- If a single DML statement fails during execution, only that statement is rolled back.
- The Oracle server implements an implicit savepoint.
- All other changes are retained.
- The user should terminate transactions explicitly by executing a `COMMIT` or `ROLLBACK` statement.

Read Consistency

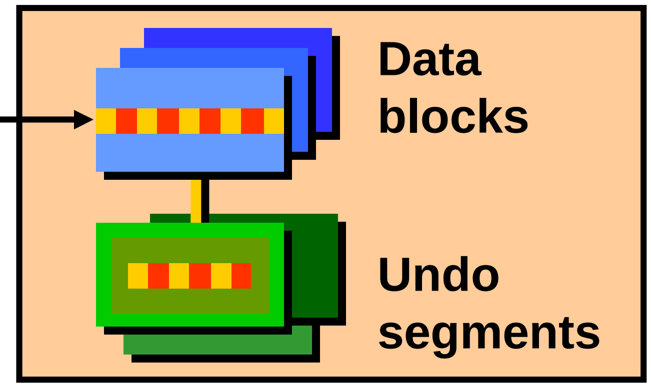
- Read consistency guarantees a consistent view of the data at all times.
- Changes made by one user do not conflict with changes made by another user.
- Read consistency ensures that on the same data:
  - Readers do not wait for writers
  - Writers do not wait for readers

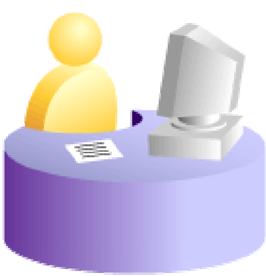
Implementation of Read Consistency

User A



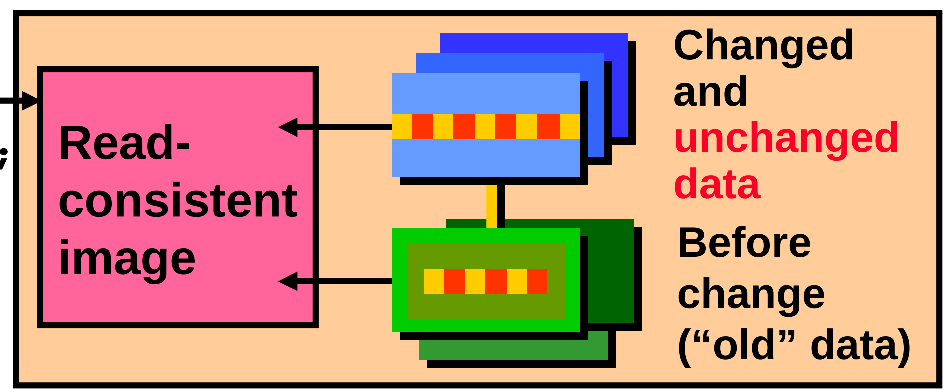
```
UPDATE  employees
SET     salary = 7000
WHERE   last_name = 'Grant';
```





User B

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



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
`COMMIT`	Makes all pending changes permanent
`SAVEPOINT`	Is used to roll back to the savepoint marker
`ROLLBACK`	Discards all pending data changes

Practice 8: Overview

This practice covers the following topics:

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

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
- Understand how constraints are created at the time of table creation
- Describe how schema objects work

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 names to objects

Naming Rules

Table names and column names:

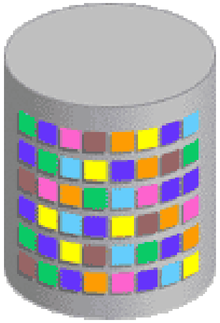
- Must begin with a letter
- Must be 1–30 characters long
- Must contain only A–Z, a–z, 0–9, _, $, and #
- Must not duplicate the name of another object owned by the same user
- Must not be an Oracle server reserved word

`CREATE TABLE` Statement

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

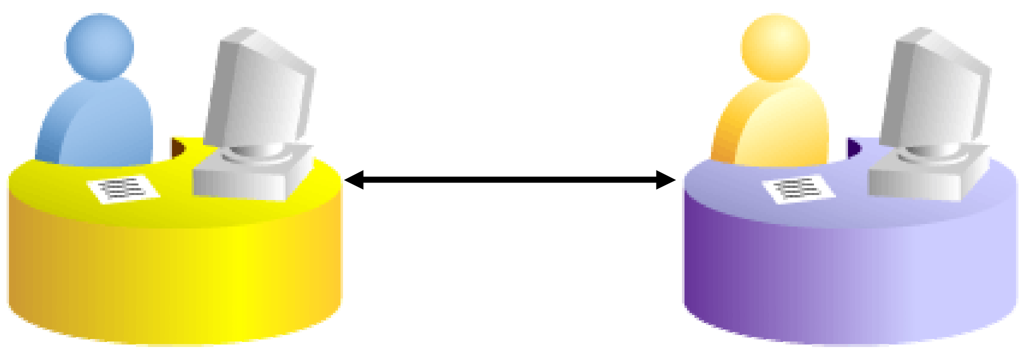
```
CREATE TABLE [schema.]table
          (column datatype [DEFAULT expr][, ...]);
```

- **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.




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

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

`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);
Table created.
```

Creating Tables

- Create the table.

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

- Confirm table creation.

```
DESCRIBE dept
```

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

Data Types

Data Type	Description
`VARCHAR2(`*`size`*`)`	Variable-length character data
`CHAR(`*`size`*`)`	Fixed-length character data
`NUMBER(`*`p,s`*`)`	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

Datetime Data Types

- The `TIMESTAMP` data type is an extension of the `DATE` data type.
- It stores the year, month, and day of the `DATE` data type plus hour, minute, and second values as well as the fractional second value.
- You can optionally specify the time zone.

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

```
TIMESTAMP[(fractional_seconds_precision)]
WITH TIME ZONE
```

```
TIMESTAMP[(fractional_seconds_precision)]
WITH LOCAL TIME ZONE
```

ORACLE

Datetime Data Types

- The `INTERVAL YEAR TO MONTH` data type stores a period of time using the `YEAR` and `MONTH` datetime fields:

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

- The `INTERVAL DAY TO SECOND` data type stores a period of time in terms of days, hours, minutes, and seconds:

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

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 table is created
  – After the table has been created
- 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:**

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

- **Table-level constraint:**

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

Defining Constraints

- Column-level constraint:

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

1

- 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	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID
100	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000	90
101	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000	90
102	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000	90
103	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000	60
104	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000	60
178	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	7000	
200	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	4400	10

...

20 rows selected.

NOT NULL constraint (No row can contain a null value for this column.)

NOT NULL constraint

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

ORACLE

`UNIQUE` Constraint

EMPLOYEES

`UNIQUE` constraint

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

...

`INSERT INTO`

208	Smith	JSMITH	**Allowed**
209	Smith	JSMITH	**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));
```

ORACLE

PRIMARY KEY Constraint

DEPARTMENTS

PRIMARY KEY

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

...

INSERT INTO

Not allowed (null value)

	Public Accounting		1400
50	Finance	124	1500

Not allowed (50 already exists)

FOREIGN KEY Constraint

DEPARTMENTS

PRIMARY KEY

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

...

EMPLOYEES

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
102	De Haan	90
103	Hunold	60
104	Ernst	60
107	Lorentz	60

FOREIGN KEY

...

INSERT INTO

200	Ford	9	Not allowed (9 does not exist)
201	Ford	60	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));
```

ORACLE

`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)
    , department_id  NUMBER(4)
        CONSTRAINT     emp_dept_fk       REFERENCES
           departments (department_id));
```

Violating Constraints

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

```
UPDATE employees
       *
ERROR at line 1:
ORA-02291: integrity constraint (HR.EMP_DEPT_FK)
violated - parent key not found
```

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;
```

```
DELETE FROM departments
            *
ERROR at line 1:
ORA-02292: integrity constraint (HR.EMP_DEPT_FK)
violated - child record found
```

Creating a Table by 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 by Using a Subquery

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

```
DESCRIBE dept80
```

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

`ALTER TABLE` Statement

Use the `ALTER TABLE` statement to:

- Add a new column
- Modify an existing column
- Define a default value for the new column
- Drop a column

Dropping a Table

- All data and structure in the table are deleted.
- Any pending transactions are committed.
- All indexes are dropped.
- All constraints are dropped.
- You *cannot* roll back the `DROP TABLE` statement.

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

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
- Understand how constraints are created at the time of table creation
- Describe how schema objects work

Practice 9: 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
- Dropping tables

10

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

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 names to objects

What Is a View?

EMPLOYEES table

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALA
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	240
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	170
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	170
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	90
104	Bruce	Ernst	BERNST	590.423.4868	21-MAY-91	IT_PROG	60
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	42
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	58
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	35
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	31
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	26
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	JUL-98	ST_CLERK	25
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	JAN-00	SA_MAN	105
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	MAY-96	SA_REP	110
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	MAR-98	SA_REP	86
178	Kimberely	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	70
200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	44
201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-96	MK_MAN	130
202	Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK_REP	60
205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	120
206	William	Gietz	WGIETZ	515.123.8181	07-JUN-94	AC_ACCOUNT	83

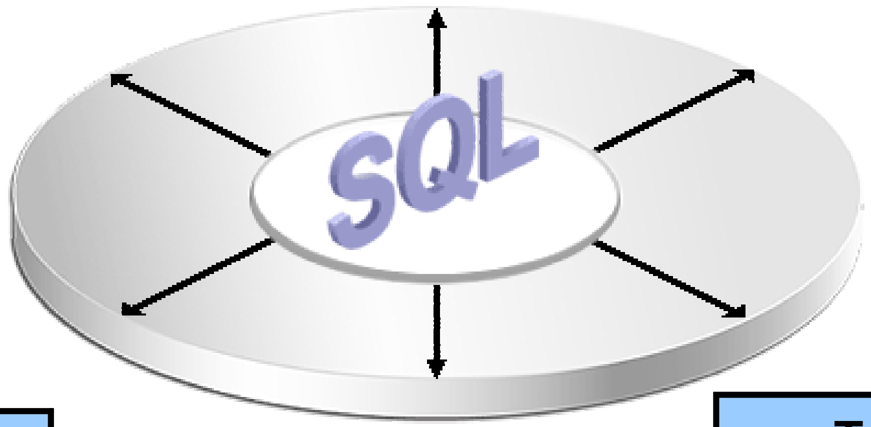
20 rows selected.

EMPLOYEE_ID	LAST_NAME	SALARY
149	Zlotkey	10500
174	Abel	11000
176	Taylor	8600

Advantages of Views

To restrict data access

To make complex queries easy



To provide data independence

To present different views of the same data

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 employees in department 80:

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

- Describe the structure of the view by using the *i*SQL*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;
View created.
```

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

Retrieving Data from a View

```
SELECT *
FROM   salvu50;
```

ID_NUMBER	NAME	ANN_SALARY
124	Mourgos	69600
141	Rajs	42000
142	Davies	37200
143	Matos	31200
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;
View created.
```

- 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;
View created.
```

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 ;
View created.
```

- Any attempt to change 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 ;
View created.
```

Removing a View

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

```
DROP VIEW view;
```

```
DROP VIEW empvu80;
View dropped.
```

Practice 10: 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**

ORACLE

Sequences

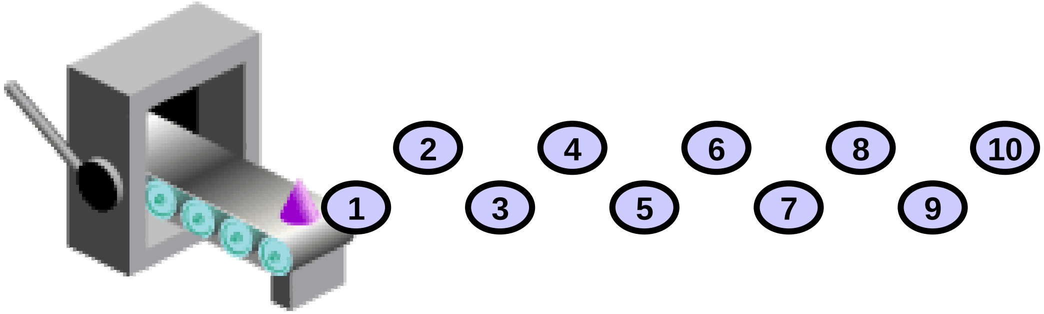
Object	Description
Table	Basic unit of storage; composed of rows
View	Logically represents subsets of data from one or more tables
Sequence	Generates numeric values
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

ORACLE

Sequences

A sequence:

- Can automatically generate unique numbers
- Is a sharable 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;
Sequence created.
```

`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 row created.
```

- 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;
Sequence altered.
```

Guidelines for Modifying a Sequence

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

```
DROP SEQUENCE dept_deptid_seq;
Sequence dropped.
```

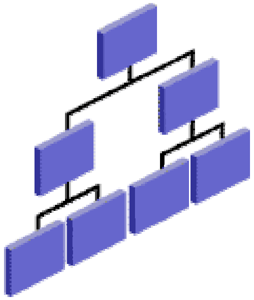
Indexes

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

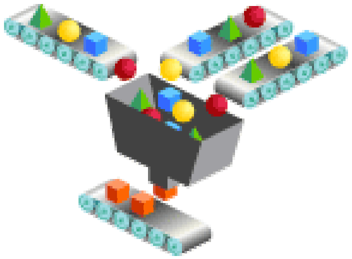
An index:

- Is a schema object
- Can be used by the Oracle server to speed up the retrieval of rows by using a pointer
- Can reduce disk 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 INDEX index
ON table (column[, column]...);
```

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

```
CREATE INDEX emp_last_name_idx
ON           employees(last_name);
Index created.
```

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 `UPPER_LAST_NAME_IDX` index from the data dictionary:

```
DROP INDEX emp_last_name_idx;
Index dropped.
```

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

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

Synonyms

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;
Synonym Created.
```

- Drop a synonym:

```
DROP SYNONYM d_sum;
Synonym dropped.
```

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 query retrieval speed
- Use synonyms to provide alternative names for objects

ORACLE

Practice 10: Overview of Part 2

This practice covers the following topics:

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

Managing Objects with Data Dictionary Views

Objectives

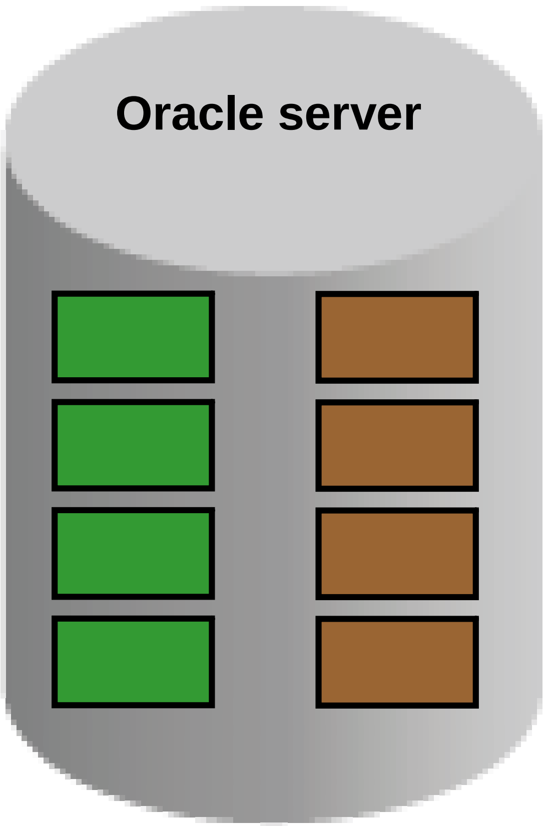
After completing this lesson, you should be able to do the following:

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

The Data Dictionary

Tables containing business data:

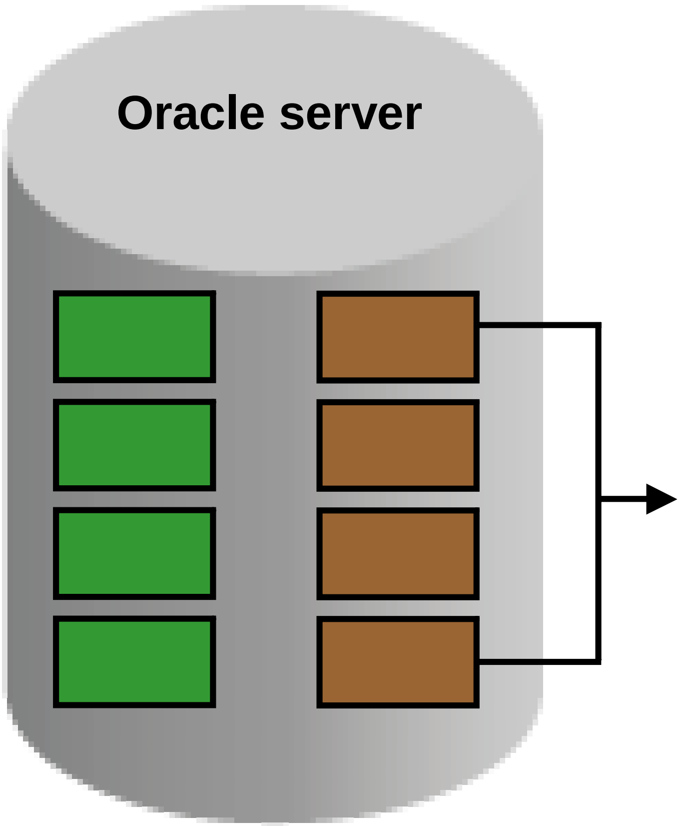
EMPLOYEES
DEPARTMENTS
LOCATIONS
JOB_HISTORY
...



Data dictionary views:

DICTIONARY
USER_OBJECTS
USER_TABLES
USER_TAB_COLUMNS
...

Data Dictionary Structure



Consists of:

– Base tables
– User-accessible views

Data Dictionary Structure

View naming convention:

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

How to Use the Dictionary Views

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

```
DESCRIBE DICTIONARY
```

Name	Null?	Type
TABLE_NAME		VARCHAR2(30)
COMMENTS		VARCHAR2(4000)

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

TABLE_NAME	COMMENTS
USER_OBJECTS	Objects owned by the user

`USER_OBJECTS` and `ALL_OBJECTS` Views

`USER_OBJECTS`:

- Query `USER_OBJECTS` to see all of the objects that are owned by you
- Is a useful way to obtain a listing of all object names and types in your schema, plus the following information:
  - Date created
  - Date of last modification
  - Status (valid or invalid)

`ALL_OBJECTS`:

- Query `ALL_OBJECTS` to see all objects to which you have access

ORACLE

USER_OBJECTS View

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

OBJECT_NAME	OBJECT_TYPE	CREATED	STATUS
REG_ID_PK	INDEX	10-DEC-03	VALID
...			
DEPARTMENTS_SEQ	SEQUENCE	10-DEC-03	VALID
REGIONS	TABLE	10-DEC-03	VALID
LOCATIONS	TABLE	10-DEC-03	VALID
DEPARTMENTS	TABLE	10-DEC-03	VALID
JOB_HISTORY	TABLE	10-DEC-03	VALID
JOB_GRADES	TABLE	10-DEC-03	VALID
EMPLOYEES	TABLE	10-DEC-03	VALID
JOBS	TABLE	10-DEC-03	VALID
COUNTRIES	TABLE	10-DEC-03	VALID
EMP_DETAILS_VIEW	VIEW	10-DEC-03	VALID

Table Information

USER_TABLES:

```
DESCRIBE user_tables
```

Name	Null?	Type
TABLE_NAME	NOT NULL	VARCHAR2(30)
TABLESPACE_NAME		VARCHAR2(30)
CLUSTER_NAME		VARCHAR2(30)
IOT_NAME		VARCHAR2(30)

```
SELECT table_name
FROM   user_tables;
```

TABLE_NAME
JOB_GRADES
REGIONS
COUNTRIES
LOCATIONS
DEPARTMENTS

...

ORACLE

Column Information

USER_TAB_COLUMNS:

```
DESCRIBE user_tab_columns
```

Name	Null?	Type
TABLE_NAME	NOT NULL	VARCHAR2(30)
COLUMN_NAME	NOT NULL	VARCHAR2(30)
DATA_TYPE		VARCHAR2(106)
DATA_TYPE_MOD		VARCHAR2(3)
DATA_TYPE_OWNER		VARCHAR2(30)
DATA_LENGTH	NOT NULL	NUMBER
DATA_PRECISION		NUMBER
DATA_SCALE		NUMBER
NULLABLE		VARCHAR2(1)
COLUMN_ID		NUMBER
DEFAULT_LENGTH		NUMBER
DATA_DEFAULT		LONG

...

Column Information

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

COLUMN_NAME	DATA_TYPE	DATA_LENGTH	DATA_PRECISION	DATA_SCALE	NUL
EMPLOYEE_ID	NUMBER	22	6	0	N
FIRST_NAME	VARCHAR2	20			Y
LAST_NAME	VARCHAR2	25			N
EMAIL	VARCHAR2	25			N
PHONE_NUMBER	VARCHAR2	20			Y
HIRE_DATE	DATE	7			N
JOB_ID	VARCHAR2	10			N
SALARY	NUMBER	22	8	2	Y
COMMISSION_PCT	NUMBER	22	2	2	Y
MANAGER_ID	NUMBER	22	6	0	Y
DEPARTMENT_ID	NUMBER	22	4	0	Y

ORACLE

Constraint Information

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

```
DESCRIBE user_constraints
```

Name	Null?	Type
OWNER	NOT NULL	VARCHAR2(30)
CONSTRAINT_NAME	NOT NULL	VARCHAR2(30)
CONSTRAINT_TYPE		VARCHAR2(1)
TABLE_NAME	NOT NULL	VARCHAR2(30)
SEARCH_CONDITION		LONG
R_OWNER		VARCHAR2(30)
R_CONSTRAINT_NAME		VARCHAR2(30)
DELETE_RULE		VARCHAR2(9)
STATUS		VARCHAR2(8)

...

Constraint Information

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

CONSTRAINT_NAME	CON	SEARCH_CONDITION	R_CONSTRAINT_NAME	DELETE_RULE	STATUS
EMP_LAST_NAME_NN	C	"LAST_NAME" IS NOT NULL			ENABLED
EMP_EMAIL_NN	C	"EMAIL" IS NOT NULL			ENABLED
EMP_HIRE_DATE_NN	C	"HIRE_DATE" IS NOT NULL			ENABLED
EMP_JOB_NN	C	"JOB_ID" IS NOT NULL			ENABLED
EMP_SALARY_MIN	C	salary > 0			ENABLED
EMP_EMAIL_UK	U				ENABLED
EMP_EMP_ID_PK	P				ENABLED
EMP_DEPT_FK	R		DEPT_ID_PK	NO ACTION	ENABLED
EMP_JOB_FK	R		JOB_ID_PK	NO ACTION	ENABLED
EMP_MANAGER_FK	R		EMP_EMP_ID_PK	NO ACTION	ENABLED

Constraint Information

```
DESCRIBE user_cons_columns
```

Name	Null?	Type
OWNER	NOT NULL	VARCHAR2(30)
CONSTRAINT_NAME	NOT NULL	VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)
COLUMN_NAME		VARCHAR2(4000)
POSITION		NUMBER

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

CONSTRAINT_NAME	COLUMN_NAME
EMP_EMAIL_UK	EMAIL
EMP_SALARY_MIN	SALARY
EMP_JOB_NN	JOB_ID
EMP_HIRE_DATE_NN	HIRE_DATE

...

ORACLE

View Information

1

```
DESCRIBE user_views
```

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

2

```
SELECT DISTINCT view_name FROM user_views;
```

VIEW_NAME
EMP_DETAILS_VIEW

3

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

TEXT
SELECT e.employee_id, e.job_id, e.manager_id, e.department_id, d.locat ion_id, l.country_id, e.first_name, e.last_name, e.salary, e.commissio n_pct, d.department_name, j.job_title, l.city, l.state_province, c.cou ntry_name, r.region_name FROM employees e, departments d, jobs j, loca tions l, countries c, regions r WHERE e.department_id = d.department_id AN D d.location_id = l.location_id AND l.country_id = c.country_id AND c.region _id = r.region_id AND j.job_id = e.job_id WITH READ ONLY

Sequence Information

```
DESCRIBE user_sequences
```

Name	Null?	Type
SEQUENCE_NAME	NOT NULL	VARCHAR2(30)
MIN_VALUE		NUMBER
MAX_VALUE		NUMBER
INCREMENT_BY	NOT NULL	NUMBER
CYCLE_FLAG		VARCHAR2(1)
ORDER_FLAG		VARCHAR2(1)
CACHE_SIZE	NOT NULL	NUMBER
LAST_NUMBER	NOT NULL	NUMBER

Sequence Information

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

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

SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	LAST_NUMBER
LOCATIONS_SEQ	1	9900	100	3300
DEPARTMENTS_SEQ	1	9990	10	280
EMPLOYEES_SEQ	1	1.0000E+27	1	207

- The `LAST_NUMBER` column displays the next available sequence number if `NOCACHE` is specified.

Synonym Information

```
DESCRIBE user_synonyms
```

Name	Null?	Type
SYNONYM_NAME	NOT NULL	VARCHAR2(30)
TABLE_OWNER		VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)
DB_LINK		VARCHAR2(128)

```
SELECT *
FROM   user_synonyms;
```

SYNONYM_NAME	TABLE_OWNER	TABLE_NAME	DB_LINK
EMP	ORA1	EMPLOYEES	

Adding Comments to a Table

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

```
COMMENT ON TABLE employees
IS 'Employee Information';
Comment created.
```

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

Summary

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

- **DICTIONARY**
- **USER_OBJECTS**
- **USER_TABLES**
- **USER_TAB_COLUMNS**
- **USER_CONSTRAINTS**
- **USER_CONS_COLUMNS**
- **USER_VIEWS**
- **USER_SEQUENCES**
- **USER_TAB_SYNONYMS**

Practice 11: Overview

This practice covers the following topics:

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

Oracle Join Syntax

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 non-equijoins
- Use outer joins to view data that generally does not meet a join condition
- Join a table to itself by using a self-join

Obtaining Data from Multiple Tables

EMPLOYEES

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
...		
202	Fay	20
205	Higgins	110
206	Gietz	110

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
60	IT	1400
80	Sales	2500
90	Executive	1700
110	Accounting	1700
190	Contracting	1700

EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
200	10	Administration
201	20	Marketing
202	20	Marketing
...		
102	90	Executive
205	110	Accounting
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.

ORACLE

Generating a Cartesian Product

`EMPLOYEES` **(20 rows)**

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90

...

202	Fay	20
205	Higgins	110
206	Gietz	110

20 rows selected.

`DEPARTMENTS` **(8 rows)**

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
60	IT	1400
80	Sales	2500
90	Executive	1700
110	Accounting	1700
190	Contracting	1700

8 rows selected.

Cartesian product:
20 x 8 = 160 rows

EMPLOYEE_ID	DEPARTMENT_ID	LOCATION_ID
100	90	1700
101	90	1700
102	90	1700
103	60	1700
104	60	1700
107	60	1700

...

160 rows selected.

Types of Joins

Oracle-proprietary joins (8*i* and earlier releases)

- Equijoin
- Non-equijoin
- Outer join
- Self-join

SQL:1999–compliant joins

- Cross join
- Natural join
- Using clause
- Full (or two-sided) outer join
- Arbitrary join condition for outer join

ORACLE

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.**

Equijoins

EMPLOYEES

EMPLOYEE_ID	DEPARTMENT_ID
200	10
201	20
202	20
124	50
141	50
142	50
143	50
144	50
103	60
104	60
107	60
149	80
174	80
176	80

…

Foreign key

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
20	Marketing
50	Shipping
50	Shipping
50	Shipping
50	Shipping
50	Shipping
60	IT
60	IT
60	IT
80	Sales
80	Sales
80	Sales

…

Primary key

Retrieving Records with Equijoins

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

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
200	Whalen	10	10	1700
201	Hartstein	20	20	1800
202	Fay	20	20	1800
124	Mourgos	50	50	1500
141	Rajs	50	50	1500
142	Davies	50	50	1500
143	Matos	50	50	1500
144	Vargas	50	50	1500

...

19 rows selected.

Additional Search Conditions Using the AND Operator

EMPLOYEES

LAST_NAME	DEPARTMENT_ID
Whalen	10
Hartstein	20
Fay	20
Mourgos	50
Rajs	50
Davies	50
Matos	50
Vargas	50
Hunold	60
Ernst	60

...

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
20	Marketing
50	Shipping
50	Shipping
50	Shipping
50	Shipping
50	Shipping
60	IT
60	IT

...

Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Use table prefixes to improve performance.
- Use column aliases to distinguish columns that have identical names but reside in different tables.

Using Table Aliases

- Use table aliases to simplify queries.
- Use table prefixes to improve performance.

```
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;
```

Joining More Than Two Tables

EMPLOYEES

LAST_NAME	DEPARTMENT_ID
King	90
Kochhar	90
De Haan	90
Hunold	60
Ernst	60
Lorentz	60
Mourgos	50
Rajs	50
Davies	50
Matos	50
Vargas	50
Zlotkey	80
Abel	80
Taylor	80

...

20 rows selected.

DEPARTMENTS

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

8 rows selected.

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.

Non-Equijoins

EMPLOYEES

LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Hunold	9000
Ernst	6000
Lorentz	4200
Mourgos	5800
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500
Zlotkey	10500
Abel	11000
Taylor	8600

…

20 rows selected.

JOB_GRADES

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

Salary in the `EMPLOYEES` table must be between lowest salary and highest salary in the `JOB_GRADES` table.

Retrieving Records with Non-Equijoins

```
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	GRA
Matos	2600	A
Vargas	2500	A
Lorentz	4200	B
Mourgos	5800	B
Rajs	3500	B
Davies	3100	B
Whalen	4400	B
Hunold	9000	C
Ernst	6000	C

...

20 rows selected.

Outer Joins

DEPARTMENTS

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

8 rows selected.

EMPLOYEES

DEPARTMENT_ID	LAST_NAME
90	King
90	Kochhar
90	De Haan
60	Hunold
60	Ernst
60	Lorentz
50	Mourgos
50	Rajs
50	Davies
50	Matos
50	Vargas
80	Zlotkey

...

20 rows selected.

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 ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Mourgos	50	Shipping
Rajs	50	Shipping
Davies	50	Shipping
Matos	50	Shipping
...		
Gietz	110	Accounting
		Contracting

20 rows selected.

ORACLE

Self-Joins

EMPLOYEES (WORKER)

EMPLOYEE_ID	LAST_NAME	MANAGER_ID
100	King	
101	Kochhar	100
102	De Haan	100
103	Hunold	102
104	Ernst	103
107	Lorentz	103
124	Mourgos	100

...

EMPLOYEES (MANAGER)

EMPLOYEE_ID	LAST_NAME
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst
107	Lorentz
124	Mourgos

...

MANAGER_ID in the WORKER table is equal to EMPLOYEE_ID in the MANAGER table.

Joining a Table to Itself

```
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
Kochhar works for King
De Haan works for King
Mourgos works for King
Zlotkey works for King
Hartstein works for King
Whalen works for Kochhar
Higgins works for Kochhar
Hunold works for De Haan
Ernst works for Hunold

...

19 rows selected.

Summary

In this appendix, you should have learned how to use joins to display data from multiple tables by using Oracle-proprietary syntax for versions 8*i* and earlier.

Practice C: Overview

This practice covers writing queries to join tables using Oracle syntax.

Using SQL*Plus

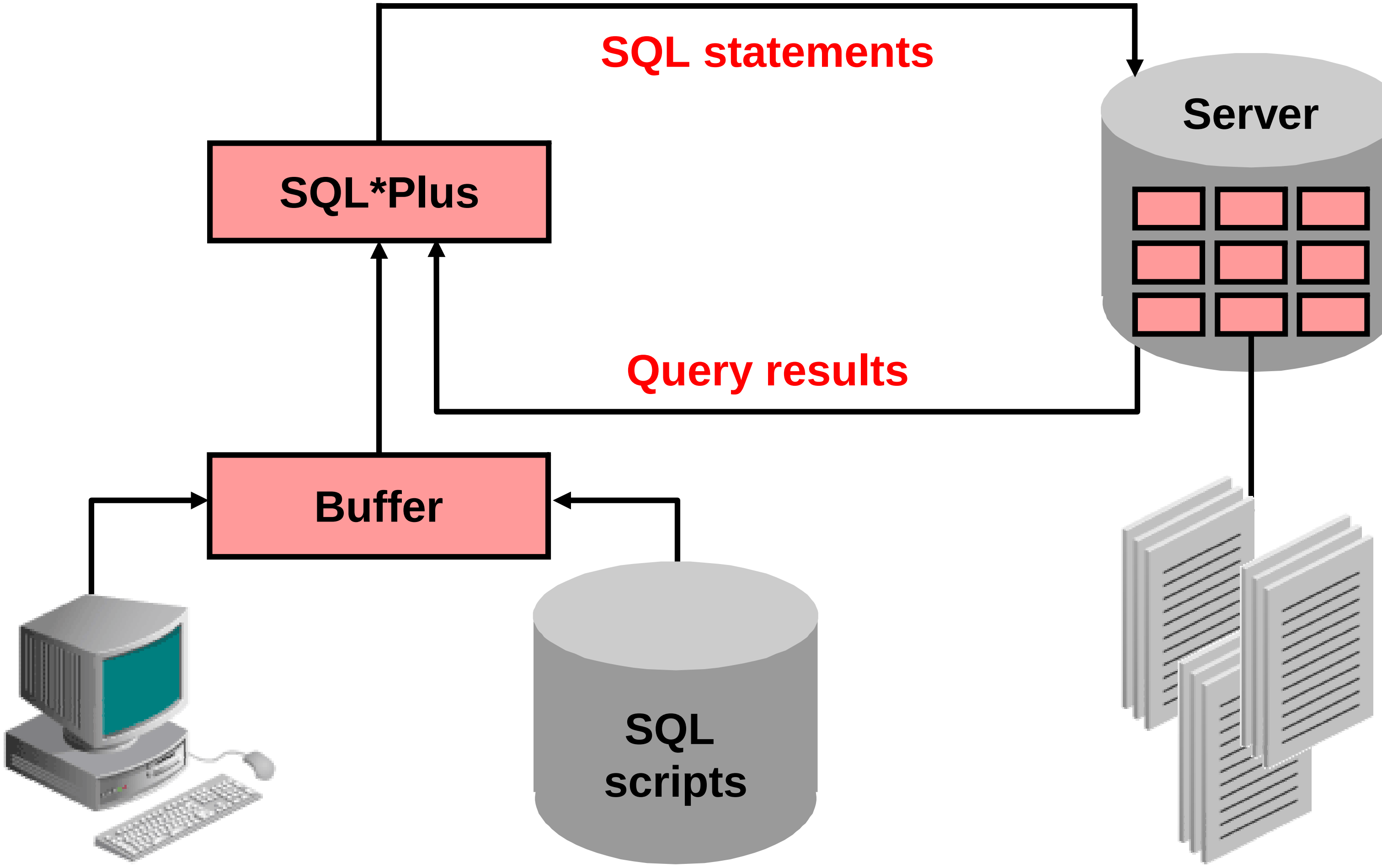
Objectives

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**

ORACLE

SQL and SQL*Plus Interaction



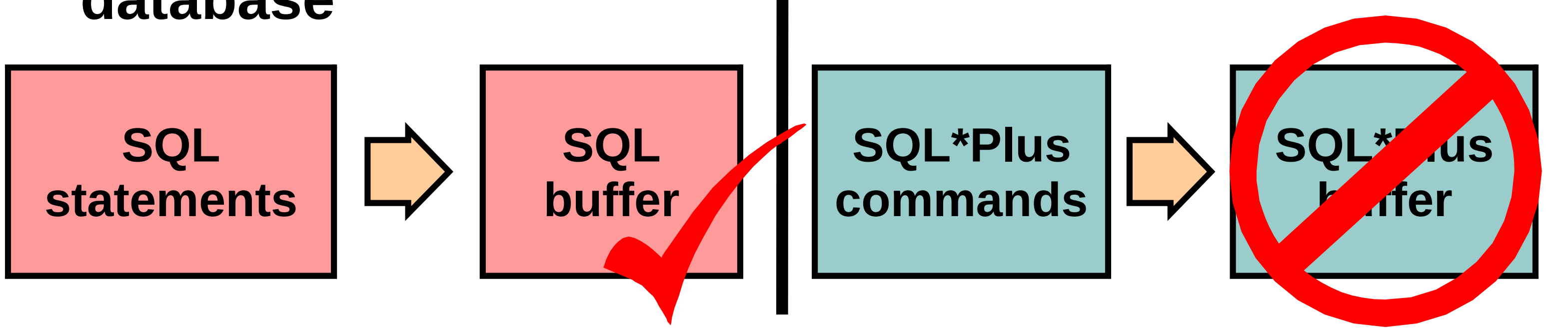
SQL Statements Versus SQL*Plus Commands

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



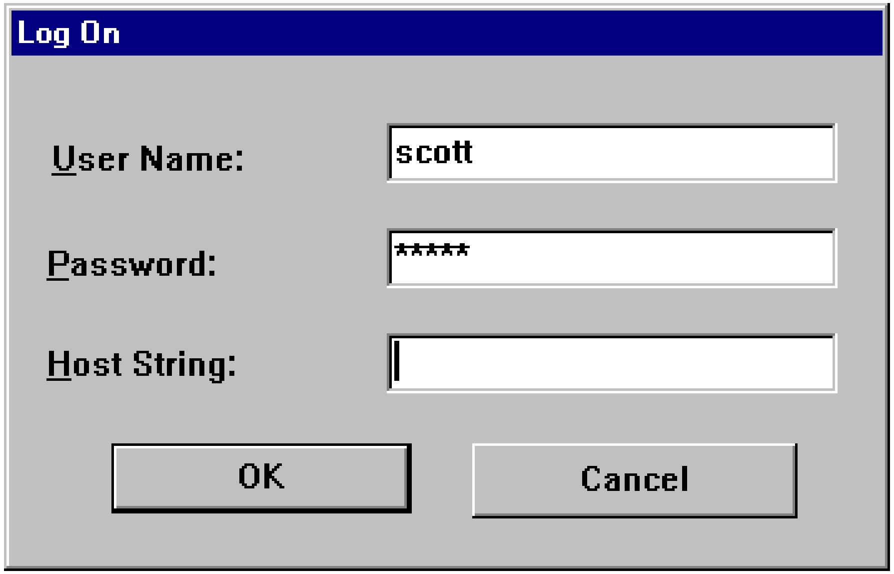
Overview of SQL*Plus

- 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.

ORACLE

Logging In to SQL*Plus

- From a Windows environment:



- From a command line:

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

Displaying Table Structure

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

```
DESC[RIBE] tablename
```

Displaying Table Structure

```
SQL> 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***

ORACLE

Using `LIST`, `n`, and `APPEND`

```
SQL> LIST
```

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

```
SQL> 1
```

```
1*  SELECT last_name
```

```
SQL> A , job_id
```

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

```
SQL> L
```

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

Using the `CHANGE` Command

```
SQL> L
```

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

```
SQL> c/employees/departments
```

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

```
SQL> L
```

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

SQL*Plus File Commands

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

Using the SAVE and START Commands

```
SQL> L
  1  SELECT last_name, manager_id, department_id
  2* FROM   employees
SQL> SAVE my_query
```

```
Created file my_query
```

```
SQL> START my_query
```

```
LAST_NAME                 MANAGER_ID DEPARTMENT_ID
------------------------- ---------- -------------
King                                            90
Kochhar                          100            90
...
20 rows selected.
```

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