
Introduction to the Oracle*9i*: SQL

Electronic Presentation

40049GC10
Production 1.0
June 2001
D33057

ORACLE®

Authors

Nancy Greenberg
Priya Nathan

Technical Contributors and Reviewers

Josephine Turner
Martin Alvarez
Anna Atkinson
Don Bates
Marco Berbeek
Andrew Brannigan
Michael Gerlach
Sharon Gray
Rosita Hanoman
Mozhe Jalali
Sarah Jones
Charbel Khouri
Christopher Lawless
Diana Lorentz
Nina Minchen
Cuong Nguyen
Daphne Nougier
Patrick Odell
Laura Pezzini
Stacey Procter
Maribel Renau
Bryan Roberts
Helen Robertson
Sunshine Salmon
Casa Sharif
Bernard Soleillant
Craig Spoonemore
Ruediger Steffan
Karla Villasenor
Andree Wheeley
Lachlan Williams

Publisher

Sheryl Domingue

Copyright © Oracle Corporation, 2000, 2001. All rights reserved.

This documentation contains proprietary information of Oracle Corporation. It is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright law. Reverse engineering of the software is prohibited. If this documentation is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend

Use, duplication or disclosure by the Government is subject to restrictions for commercial computer software and shall be deemed to be Restricted Rights software under Federal law, as set forth in subparagraph (c)(1)(ii) of DFARS 252.227-7013, Rights in Technical Data and Computer Software (October 1988).

This material or any portion of it may not be copied in any form or by any means without the express prior written permission of Oracle Corporation. Any other copying is a violation of copyright law and may result in civil and/or criminal penalties.

If this documentation is delivered to a U.S. Government Agency not within the Department of Defense, then it is delivered with "Restricted Rights," as defined in FAR 52.227-14, Rights in Data-General, including Alternate III (June 1987).

The information in this document is subject to change without notice. If you find any problems in the documentation, please report them in writing to Education Products, Oracle Corporation, 500 Oracle Parkway, Box SB-6, Redwood Shores, CA 94065. Oracle Corporation does not warrant that this document is error-free.

Oracle and all references to Oracle products are trademarks or registered trademarks of Oracle Corporation.

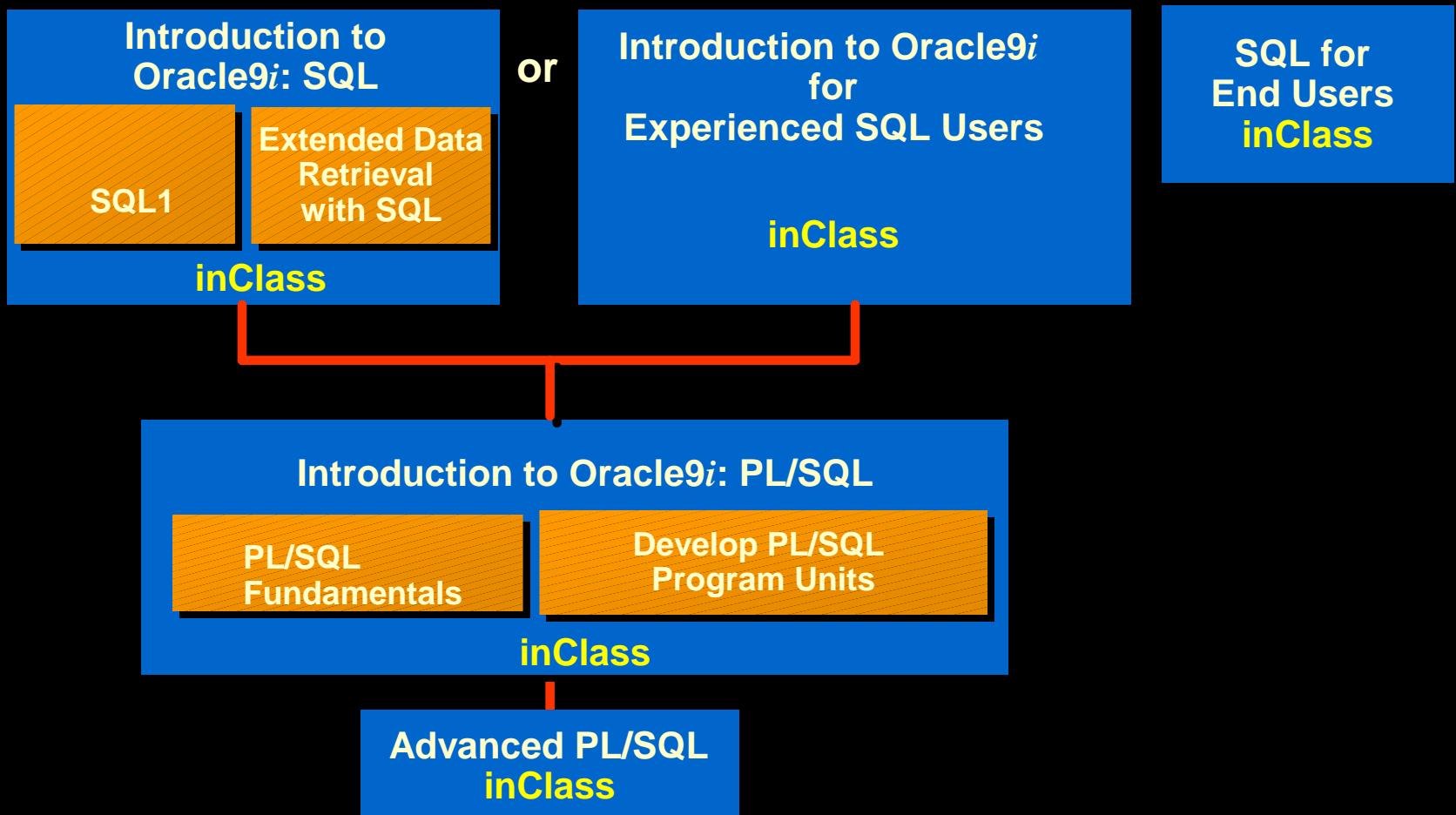
All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.

Curriculum Map

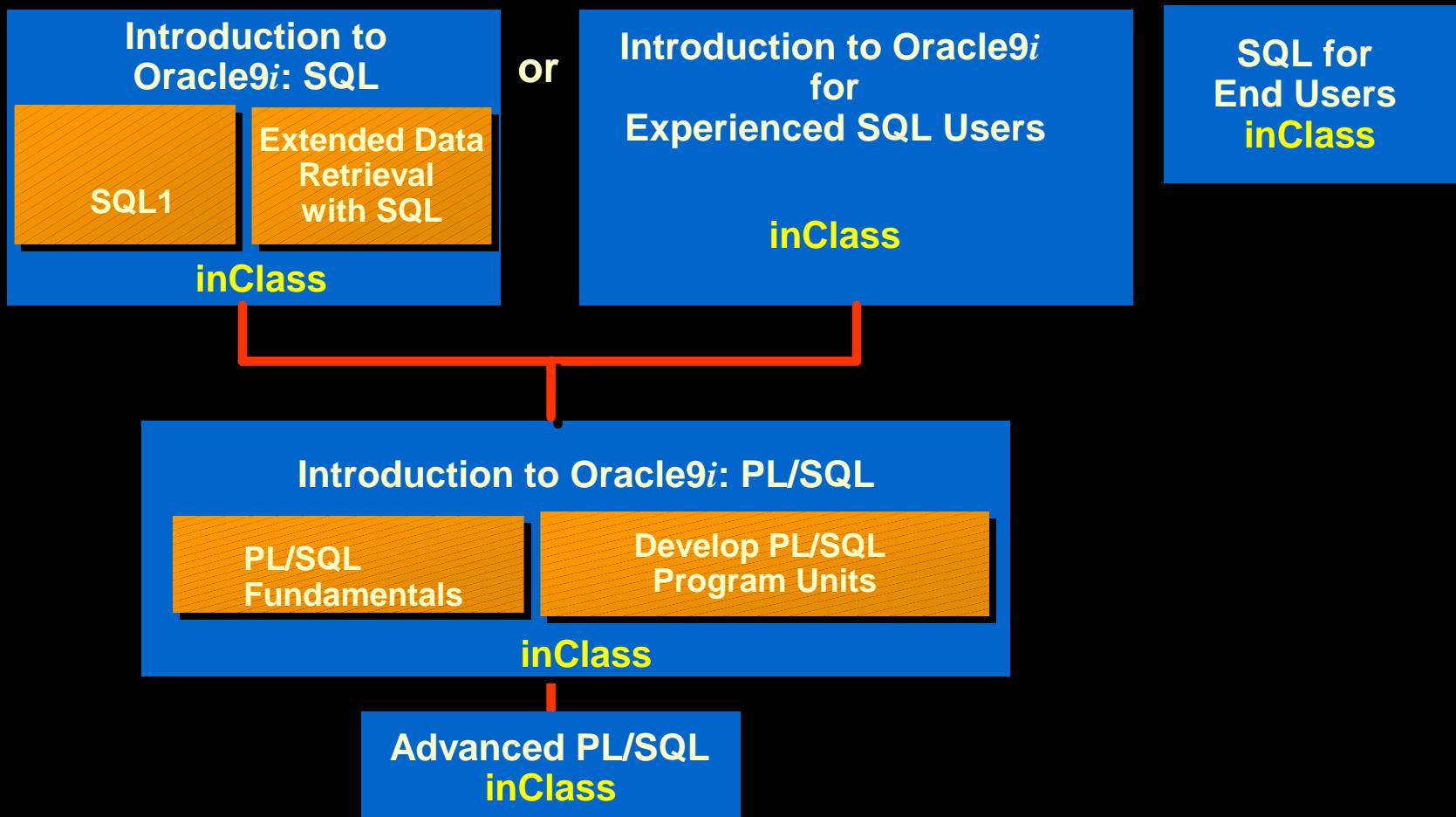
ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Languages Curriculum for Oracle9*i*



Languages Curriculum for Oracle9*i*



Introduction

Objectives

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

- **List the features of Oracle9*i***
- **Discuss the theoretical and physical aspects of a relational database**
- **Describe the Oracle implementation of the RDBMS and ORDBMS**

Oracle9*i*



Scalability

Reliability

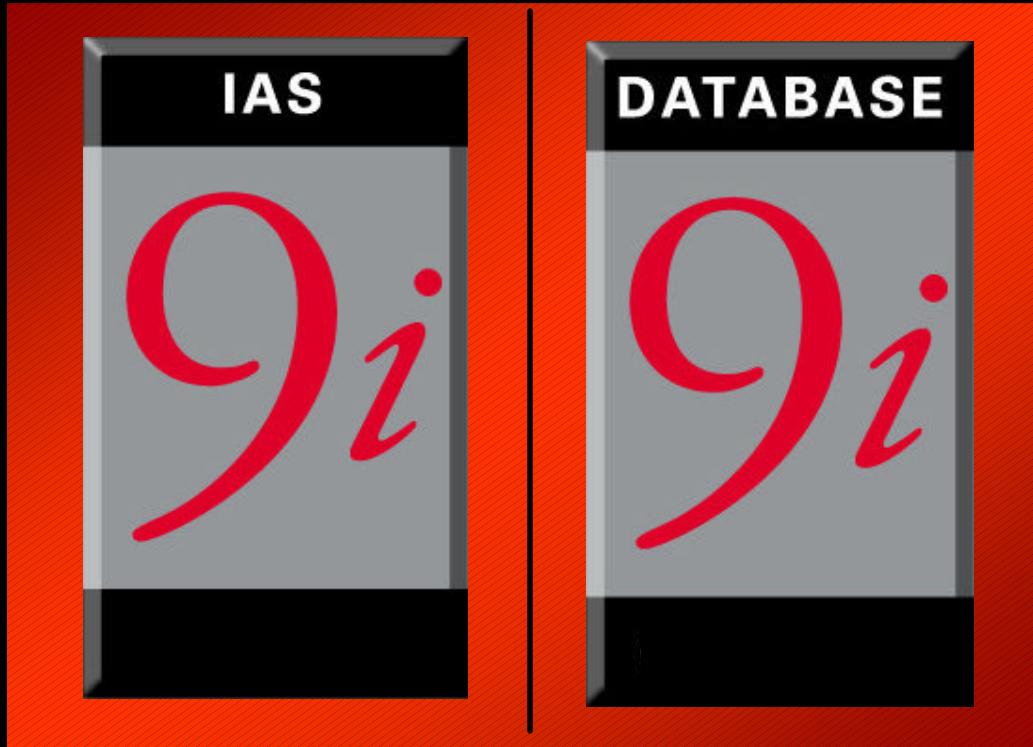
Single dev.
model

Common
skill sets

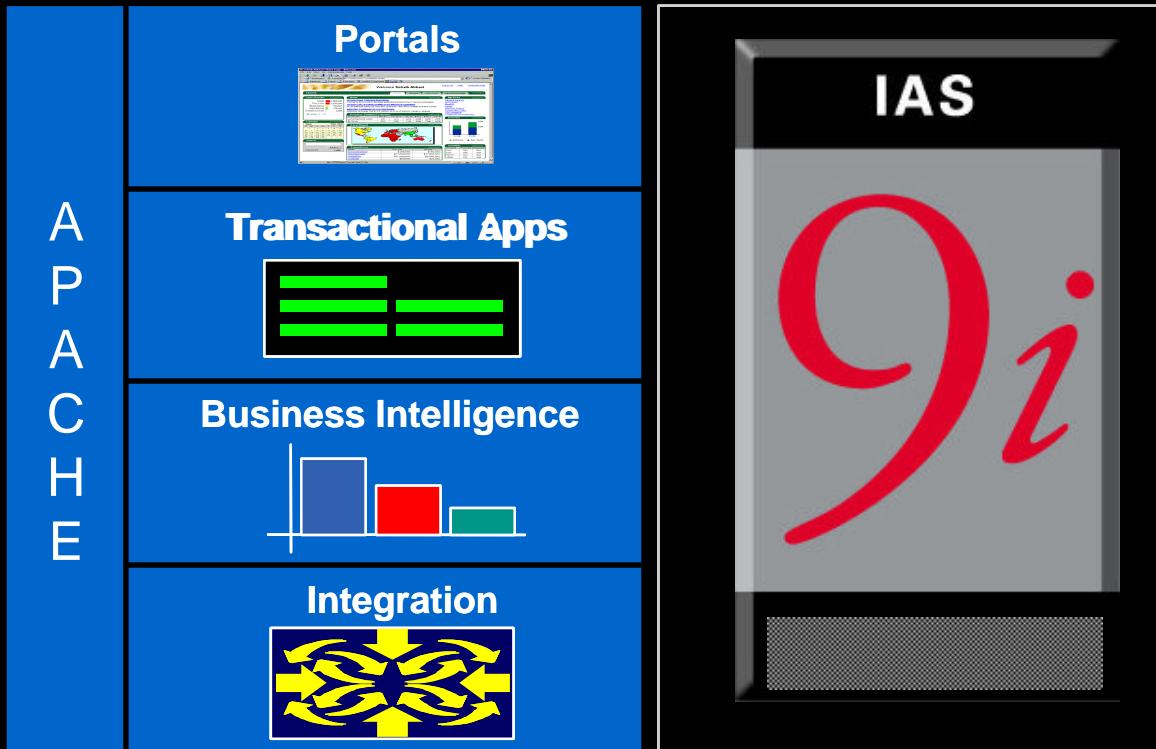
One
vendor

One mgmt.
interface

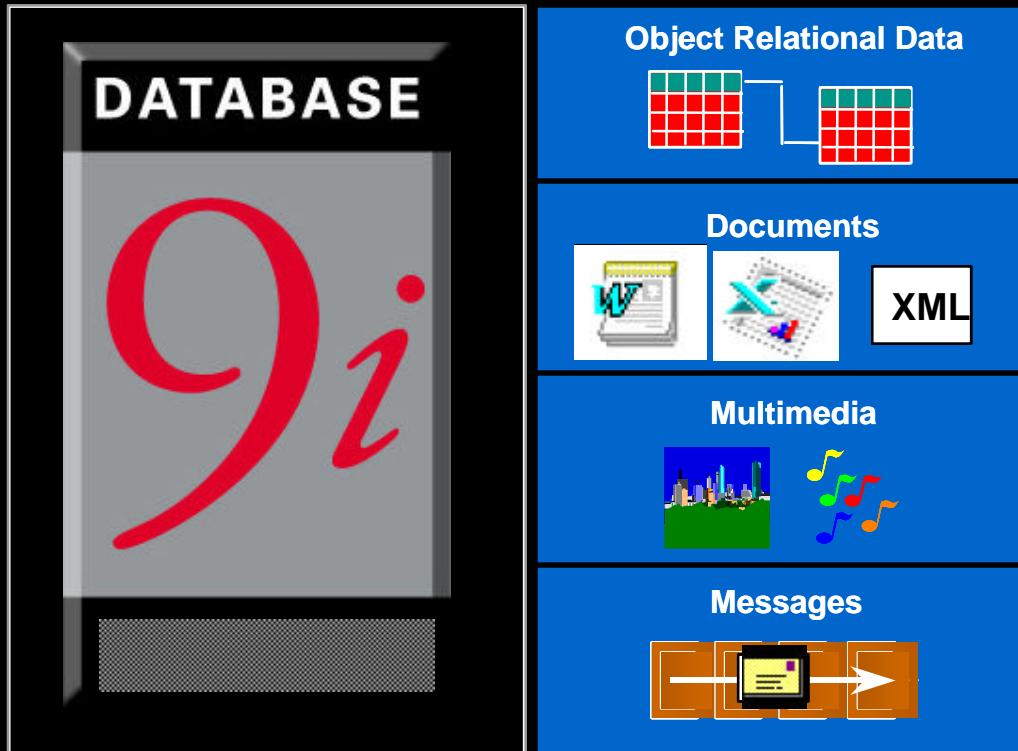
Oracle*9i*



Oracle9*i* Application Server



Oracle9i Database



Oracle9*i* Database

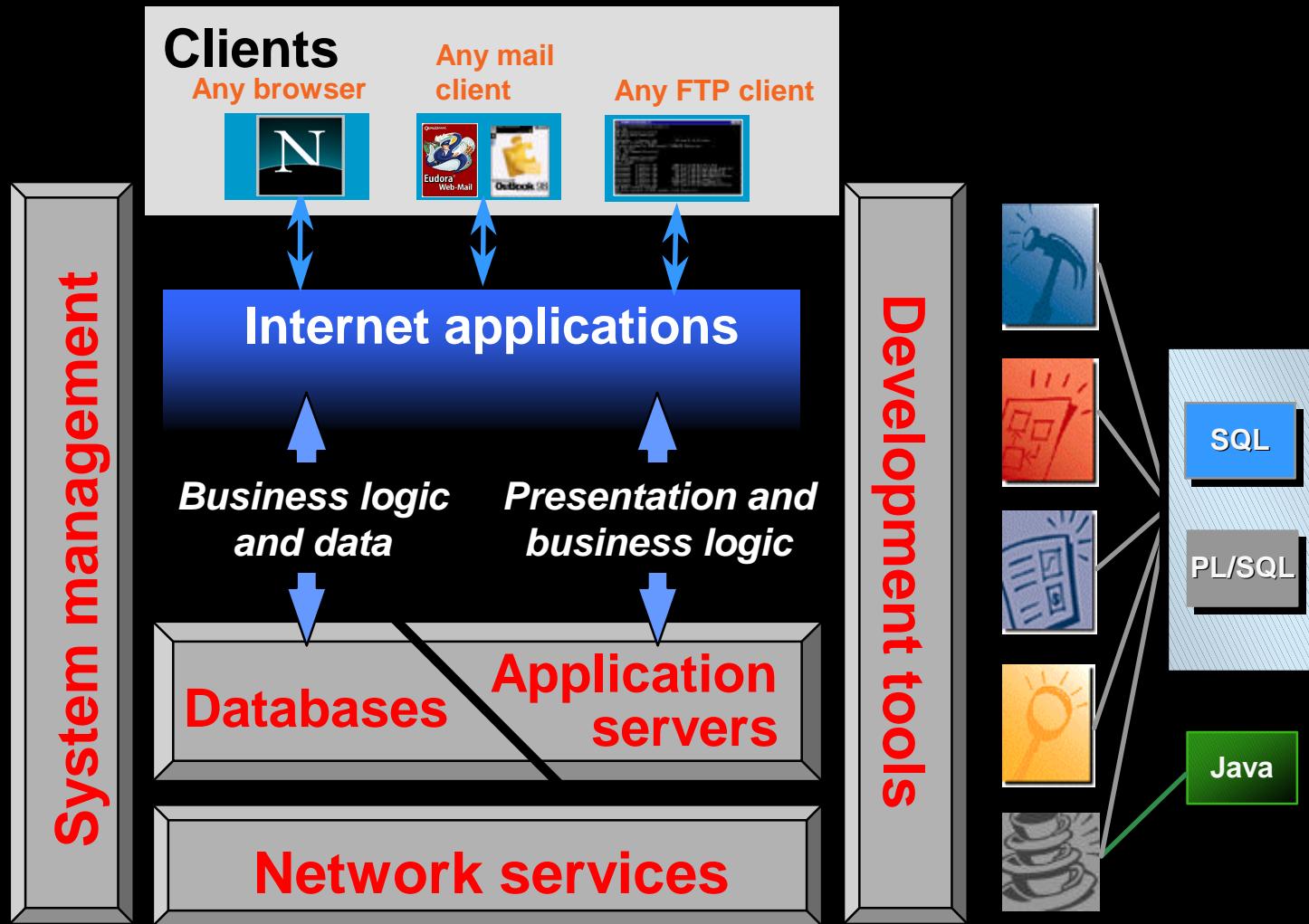


- **Performance and availability leader**
- **Richest feature set**

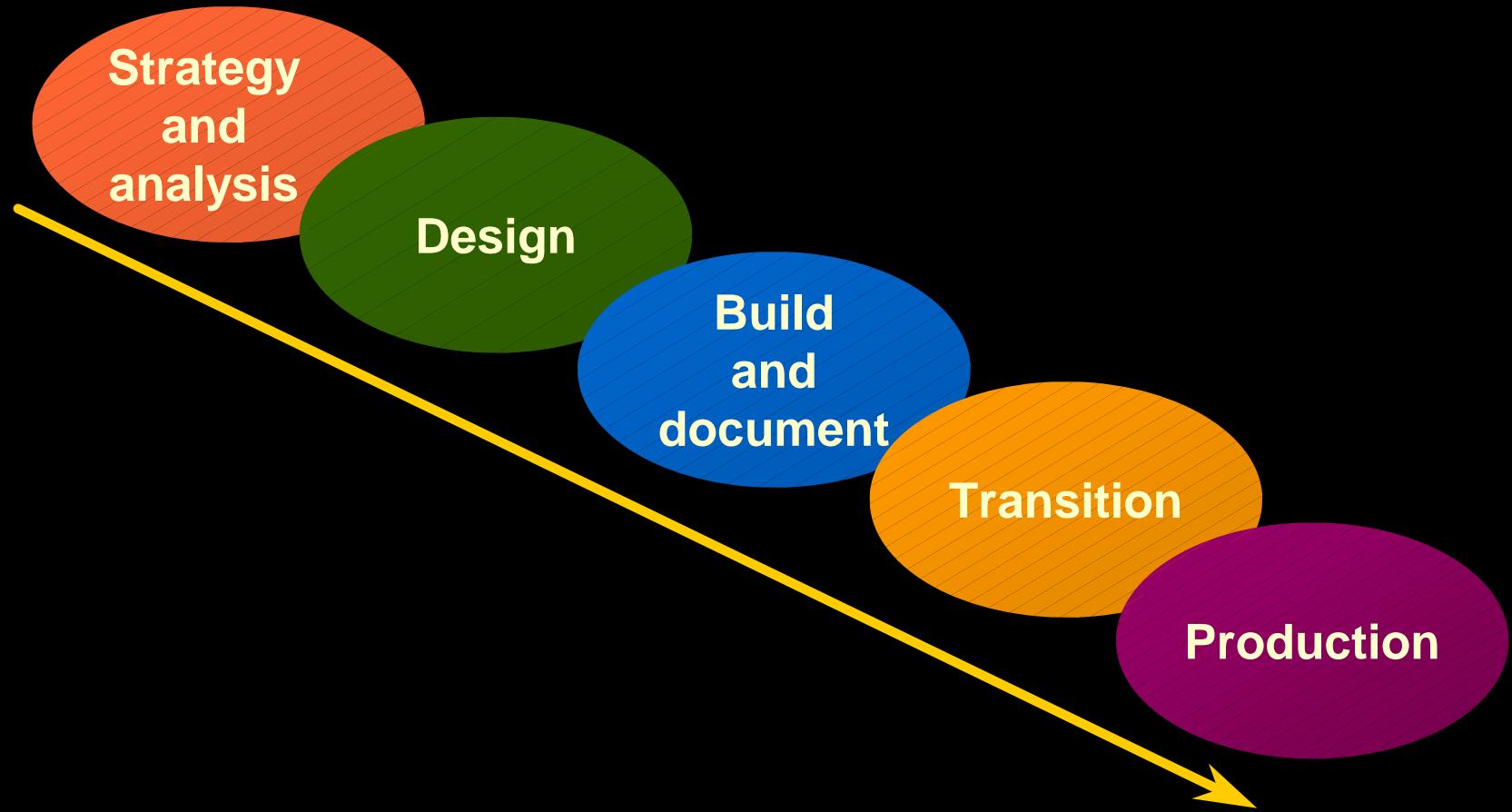
Oracle9*i*: Object Relational Database Management System

- **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



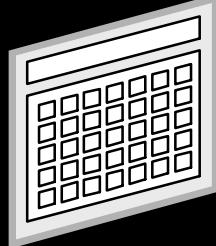
System Development Life Cycle



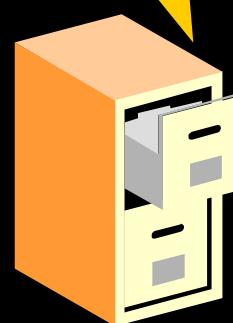
Data Storage on Different Media

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	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

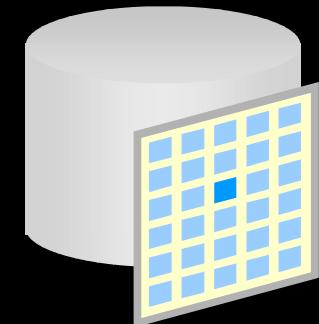
GRA	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000



Electronic
spreadsheet



Filing cabinet



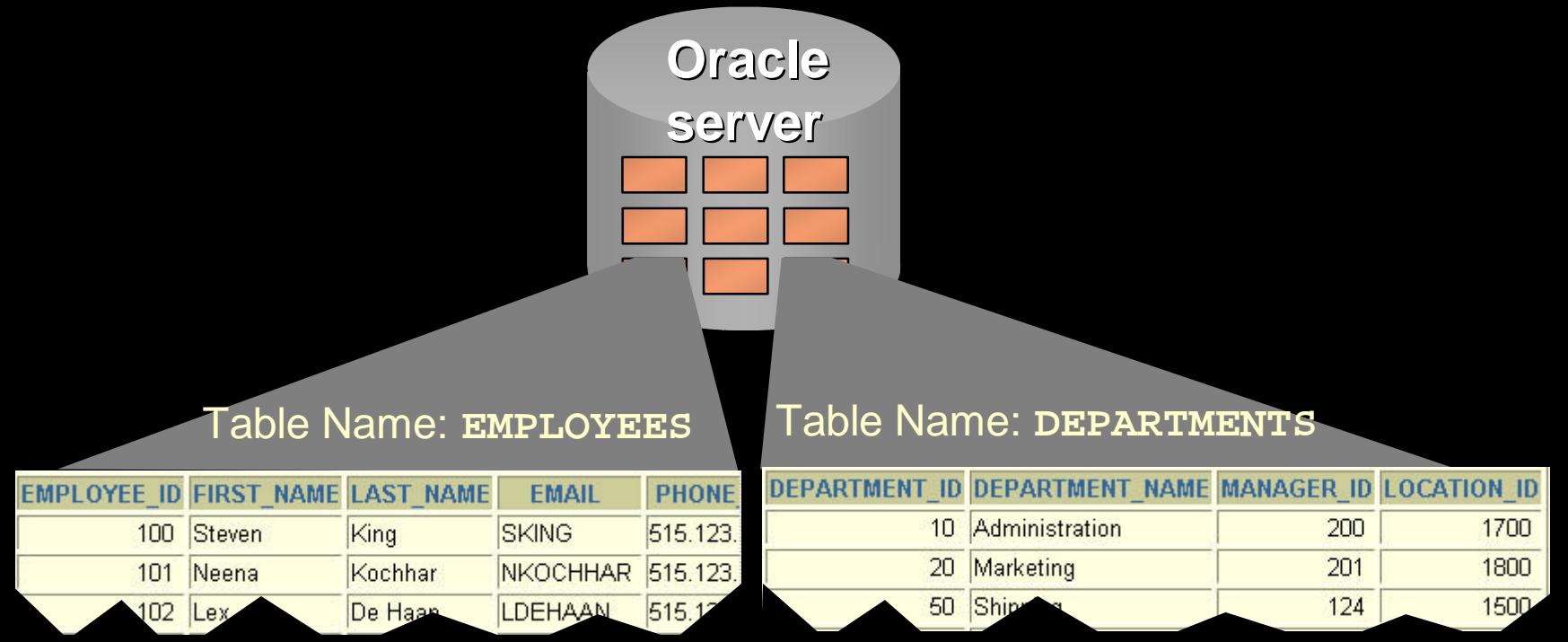
Database

Relational Database Concept

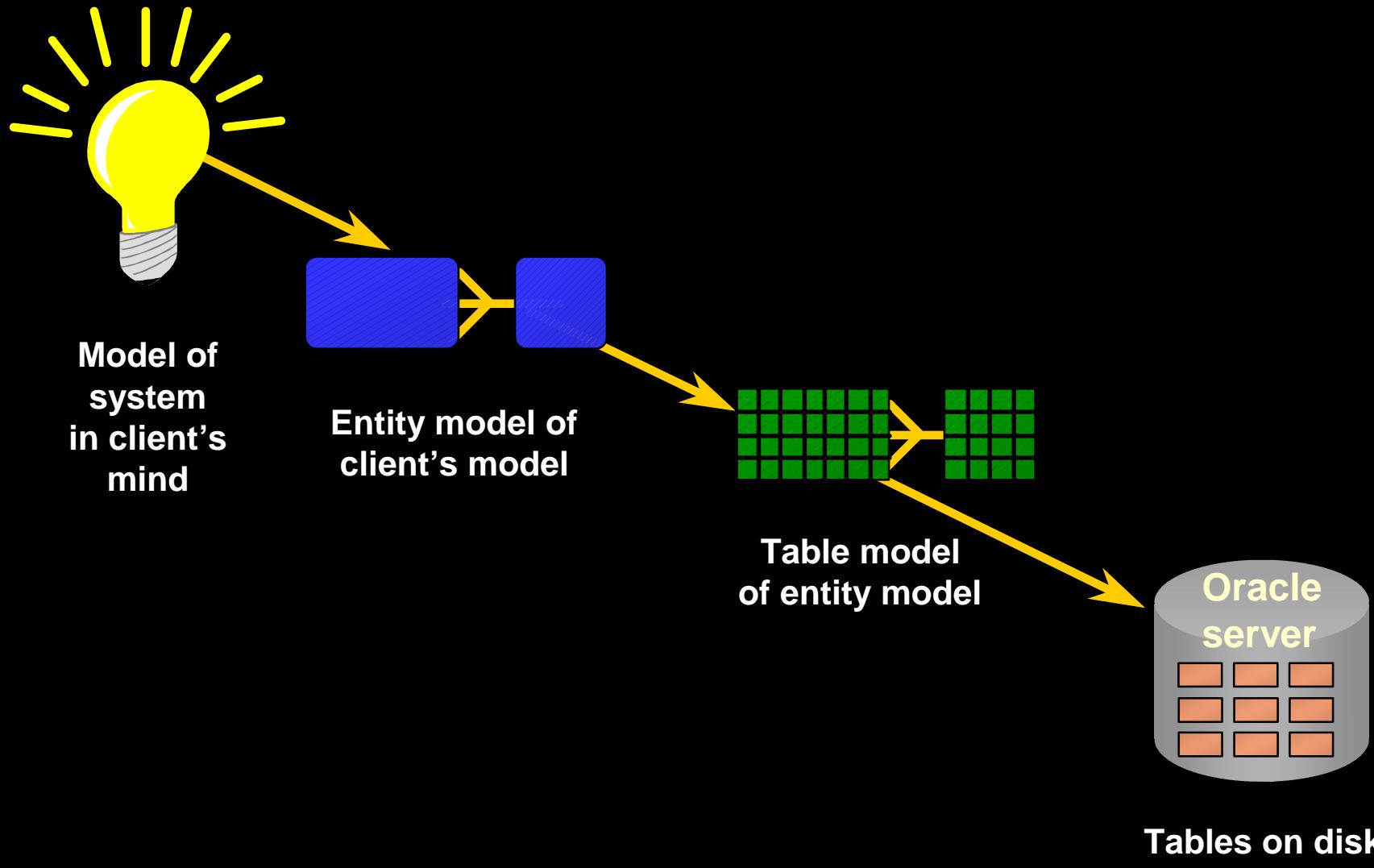
- Dr. E.F. Codd proposed the relational model for database systems in 1970.
- It is the basis for the relational database management system.
- The relational model consists of the following:
 - Collection of objects or relations
 - Set of operators to act on the relations
 - Data integrity for accuracy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

Definition of a Relational Database

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

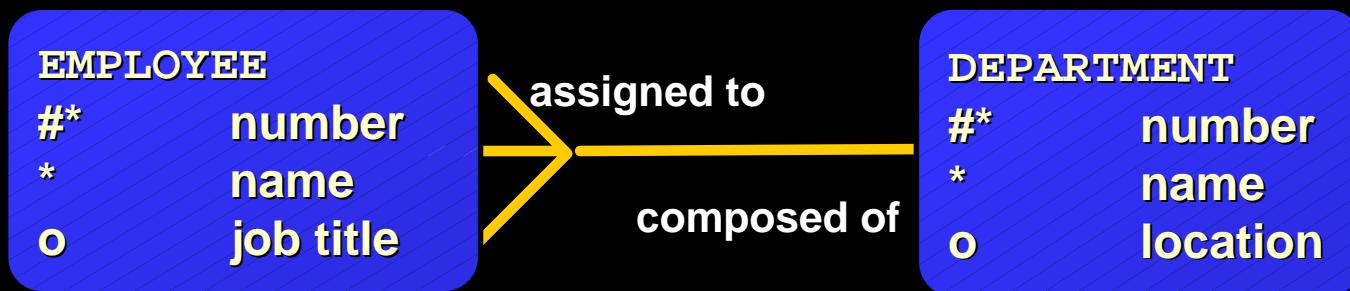


Data Models



Entity Relationship Model

- Create an entity relationship diagram from business specifications or narratives



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

Entity Relationship Modeling Conventions

Entity

Soft box

Singular, unique name

Uppercase

Synonym in parentheses

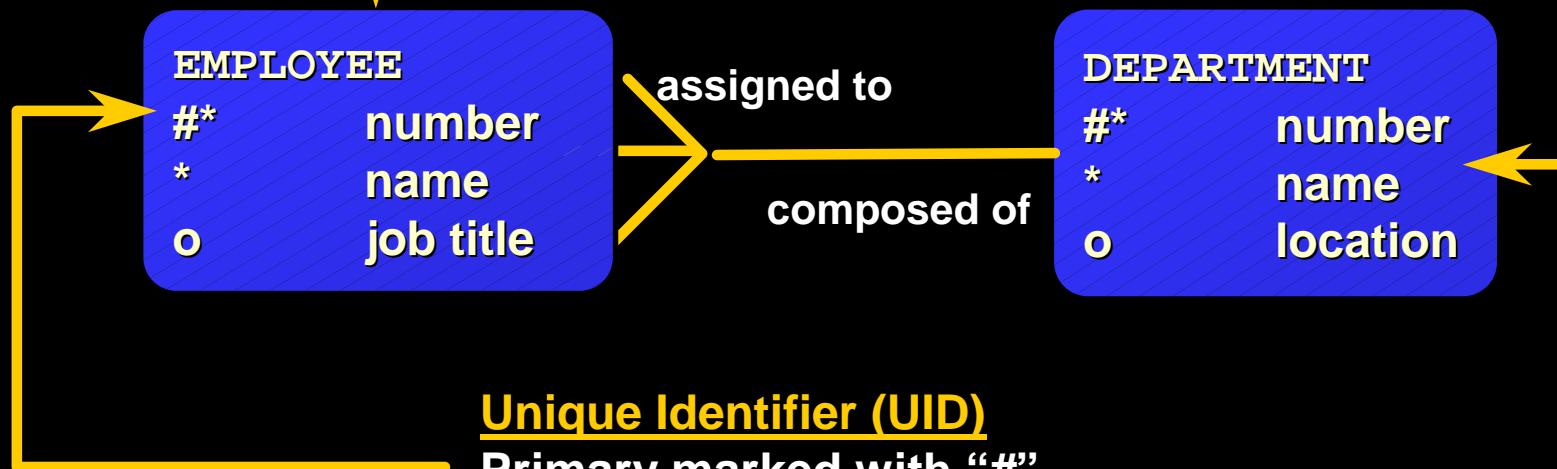
Attribute

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



Primary key

Foreign key

Table Name: DEPARTMENTS

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



Primary key

Relational Database Terminology

EMPLOYEE_ID	LAST_NAME	SALARY	COMMISSION_PCT	DEPARTMENT_ID
100	King	24000		90
101	Kochhar	17000		90
102	De Haan	17000		90
103	Hunold	9000		60
104	Ernst	6000		60
107	Lorentz	4200		60
124	Mourgos	5800		50
141	Rajs	3500		50
142	Davies	3100		50
143	Matos	2600		50
144	Vargas	2500		50
149	Zlotkey	10500	.2	80
174	Abel	11000	.3	80
178	Taylor	8600	.2	80
178	Grant	7000	.15	
200	Whalen	4400		10
201	Hartstein	13000		20
202	Fay	6000		20
205	Higgins	12000		110
206	Gietz	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 a RDBMS Using SQL

**SQL statement
is entered.**

```
SELECT department_name  
FROM   departments;
```

**Statement is sent to
Oracle Server.**

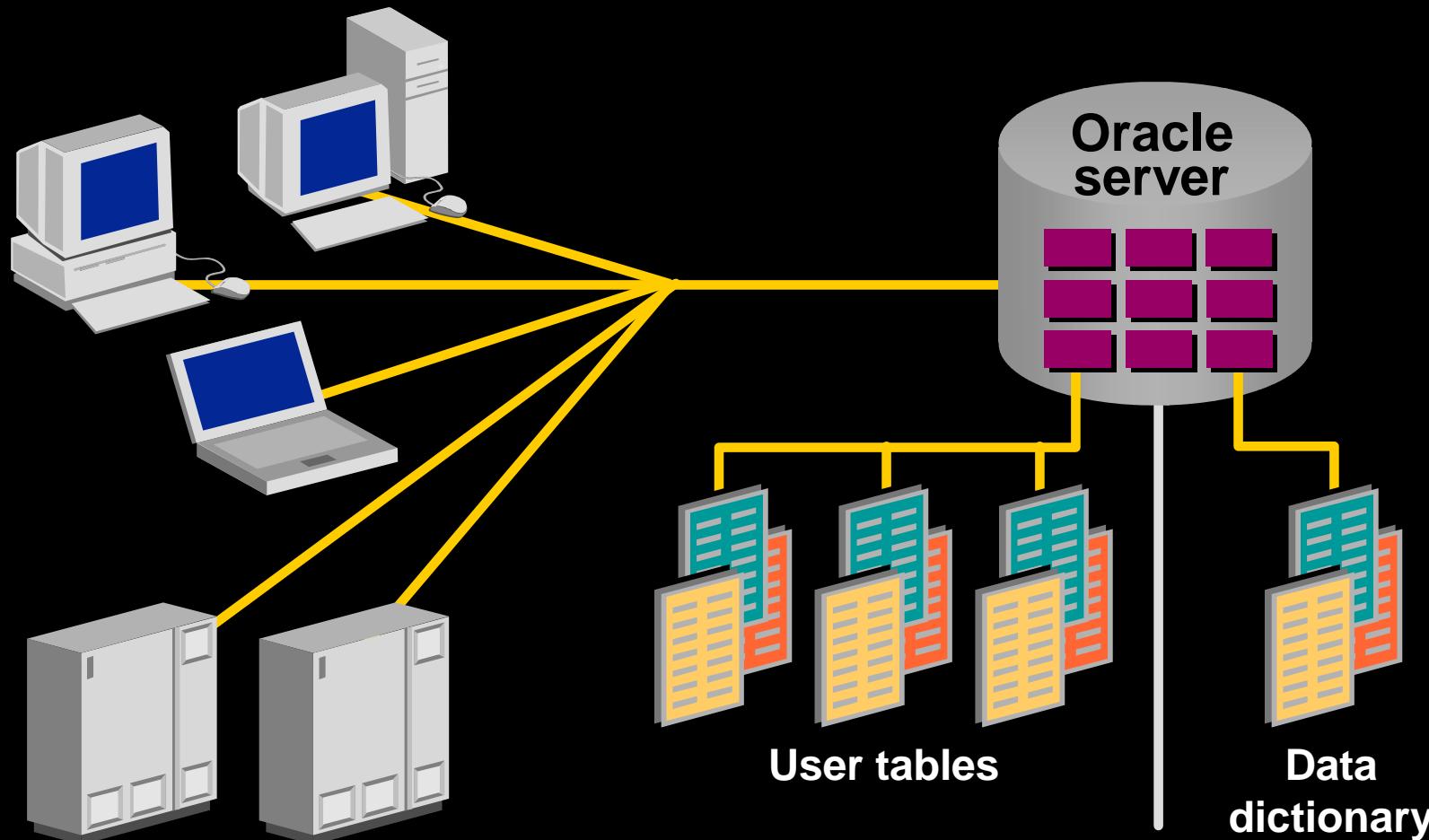
**Oracle
server**

Data is displayed.

DEPARTMENT_NAME
Administration
Marketing
Shipping
IT
Sales
Executive
Accounting
Contracting

8 rows selected.

Relational Database Management System



SQL Statements

SELECT

Data retrieval

INSERT

UPDATE

DELETE

MERGE

Data manipulation language (DML)

CREATE

ALTER

DROP

RENAME

TRUNCATE

Data definition language (DDL)

COMMIT

ROLLBACK

SAVEPOINT

Transaction control

GRANT

REVOKE

Data control language (DCL)

Tables Used in the Course

EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT
100	Steven	King	SKING	17-JUN-87	AD_PRES	24000	
101	Neena	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	17000	
102	Lex	De Haan	LDEHAAN	13-JAN-93	AD_VP	17000	
103	Alexander	Hunold	AHUNOLD	03-JAN-90	IT_PROG	9000	
104	Bruce	Ernst	BERNST	21-MAY-91	IT_PROG	6000	
107	Diana	Lorentz	DLORENTZ	07-FEB-99	IT_PROG	4200	
124	Kevin	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	5800	
141	Trenna	Rajs	TRAJS	17-OCT-95	ST_CLERK	3500	
142	Curtis	Davies	CDAVIES	29-JAN-97	ST_CLERK	3100	
143	Randall	Matos	RMATOS	15-MAR-98	ST_CLERK	2600	
144	Peter	Vargas	PVARGAS	09-JUL-98	ST_CLERK	2500	
149	Eleni	Zlotkey	EZLOTKEY	29-JAN-00	SA_MAN	10500	

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
MAY-96	SA_REP	11000
MAR-98	SA_REP	8600
MAY-98		
SEP-89	A	1000
FEB-94	B	3000
AUG-95	C	6000
JUN-94	D	10000
JUN-94	E	15000
	F	25000

JOB_GRADES

Summary

- **The Oracle*9i* Server is the database for Internet computing.**
- **Oracle*9i* 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

Writing Basic SQL SELECT Statements

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

Projection

Table 1

Selection

Table 1

Join

Table 1

Table 2

Basic SELECT Statement

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

- **SELECT identifies *what* columns**
- **FROM identifies *which* table**

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.

Column Heading Defaults

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

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
Lorentz	4200	4500

Gietz

8300

8600

20 rows selected.

Operator Precedence



- Multiplication and division take priority over addition and subtraction.
- Operators of the same priority are evaluated from left to right.
- Parentheses are used to force prioritized evaluation and to clarify statements.

Operator Precedence

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

LAST_NAME	SALARY	12*SALARY+100
King	24000	288100
Kochhar	17000	204100
De Haan	17000	204100
Hunold	9000	108100
Ernst	6000	72100
Lorentz	4200	50500

Gietz	8300	99700
-------	------	-------

20 rows selected.

Using Parentheses

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

LAST_NAME	SALARY	12*(SALARY+100)
King	24000	289200
Kochhar	17000	205200
De Haan	17000	205200
Hunold	9000	109200
Ernst	6000	73200
Lorentz	4200	51600

Gietz	8300	100800
-------	------	--------

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 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
Higgins	AC_MGR	12800	
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
	Higgins	
	Gietz	

20 rows selected.

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 is case sensitive

Using Column Aliases

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

NAME	COMM
King	
Kochhar	
Higgins	
Gietz	

20 rows selected.

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

Name	Annual Salary
King	288000
Kochhar	204000
Higgins	144000
Gietz	99600

20 rows selected.

Concatenation Operator

A concatenation operator:

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

Using the Concatenation Operator

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

Employees
KingAD_PRES
KochharAD_VP
De HaanAD_VP
HunoldIT_PROG

GietzAC_ACCOUNT

20 rows selected.

Literal Character Strings

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

Using Literal Character Strings

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

Employee Details
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

Gietz is a AC_ACCOUNT
20 rows selected.

Duplicate Rows

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

```
SELECT department_id  
FROM employees;
```

DEPARTMENT_ID
90
90
90
60
60
60
50
50

20 rows selected.



Eliminating Duplicate Rows

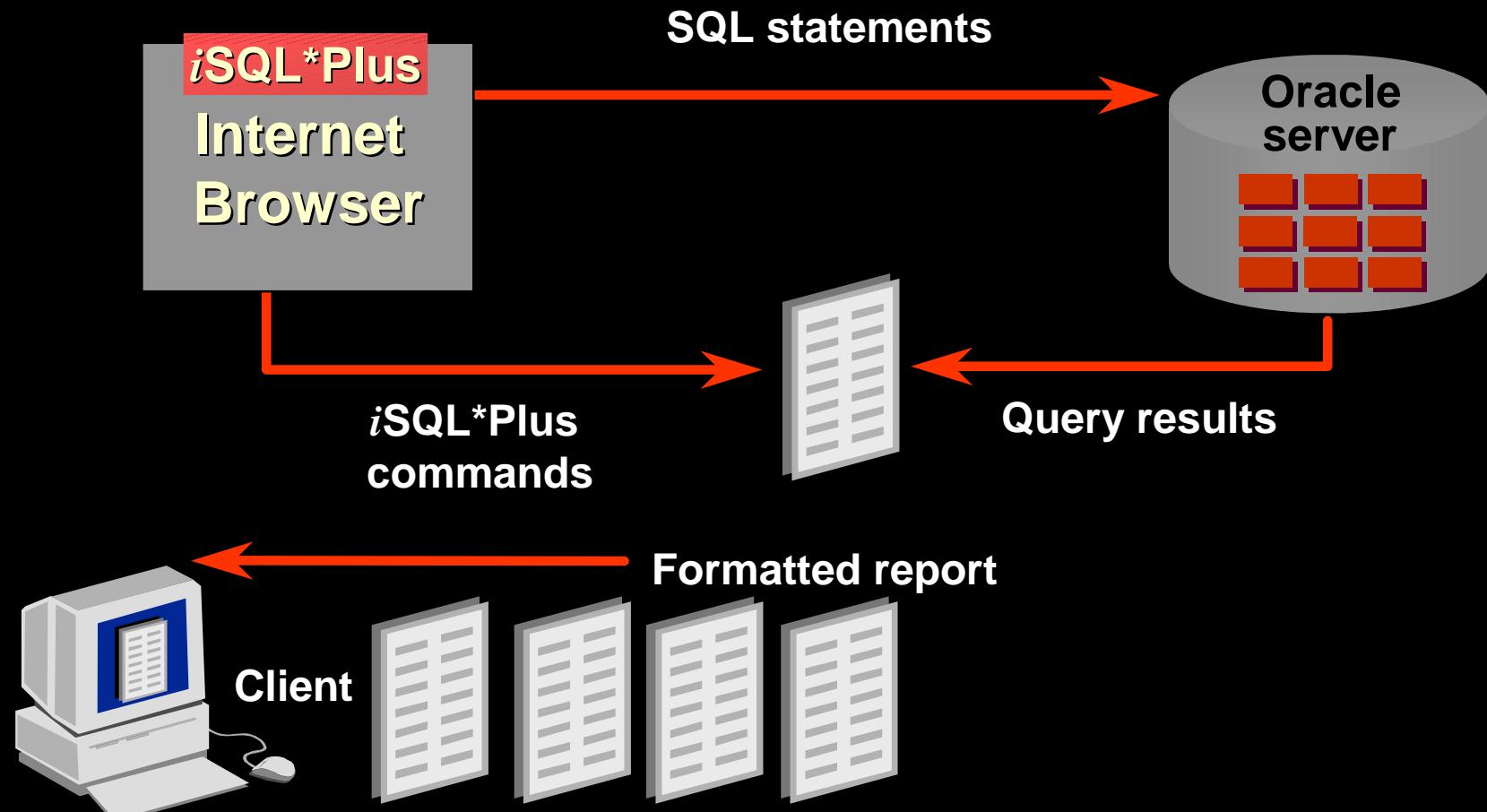
Eliminate duplicate rows by using the DISTINCT keyword in the SELECT clause.

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

DEPARTMENT_ID
10
20
50
60
80
90
110

8 rows selected.

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**

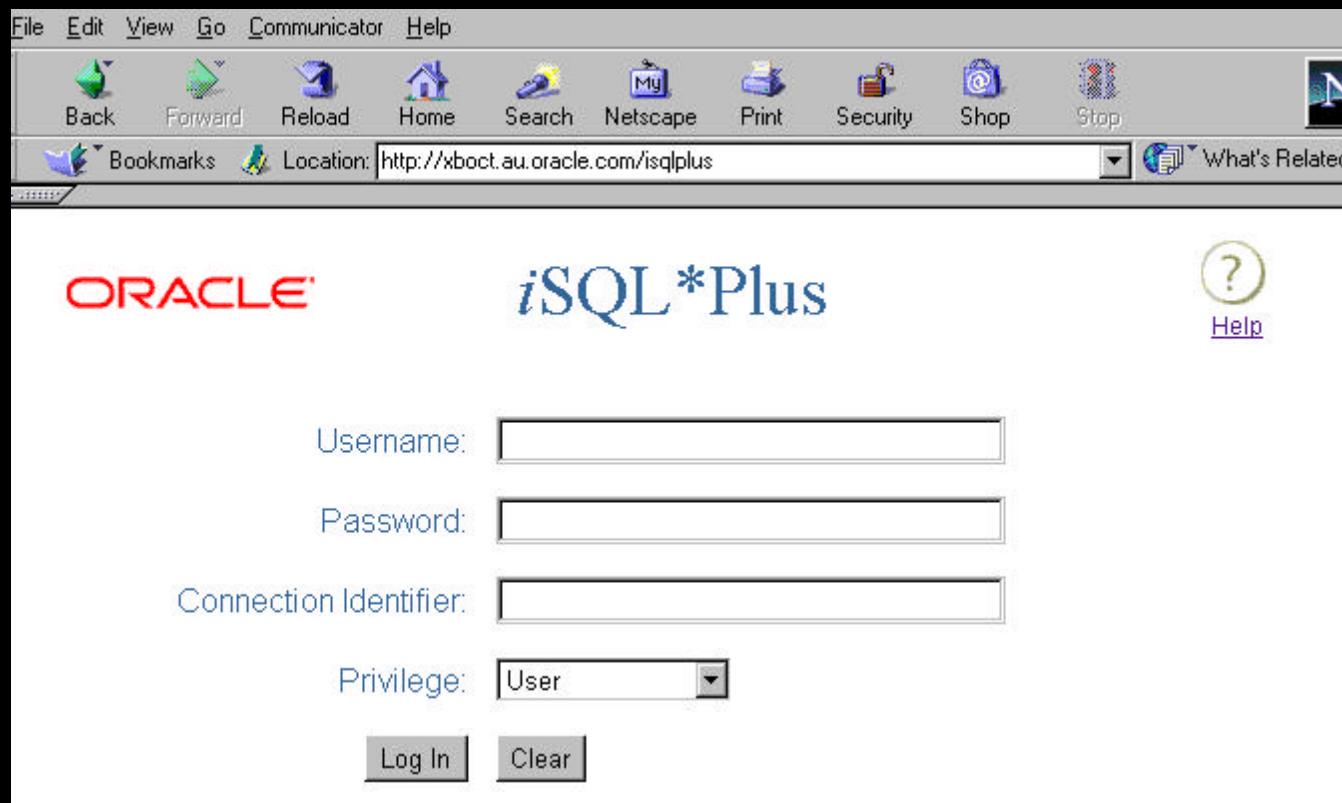
Overview of *i*SQL*Plus

After you log into *i*SQL*Plus, you can:

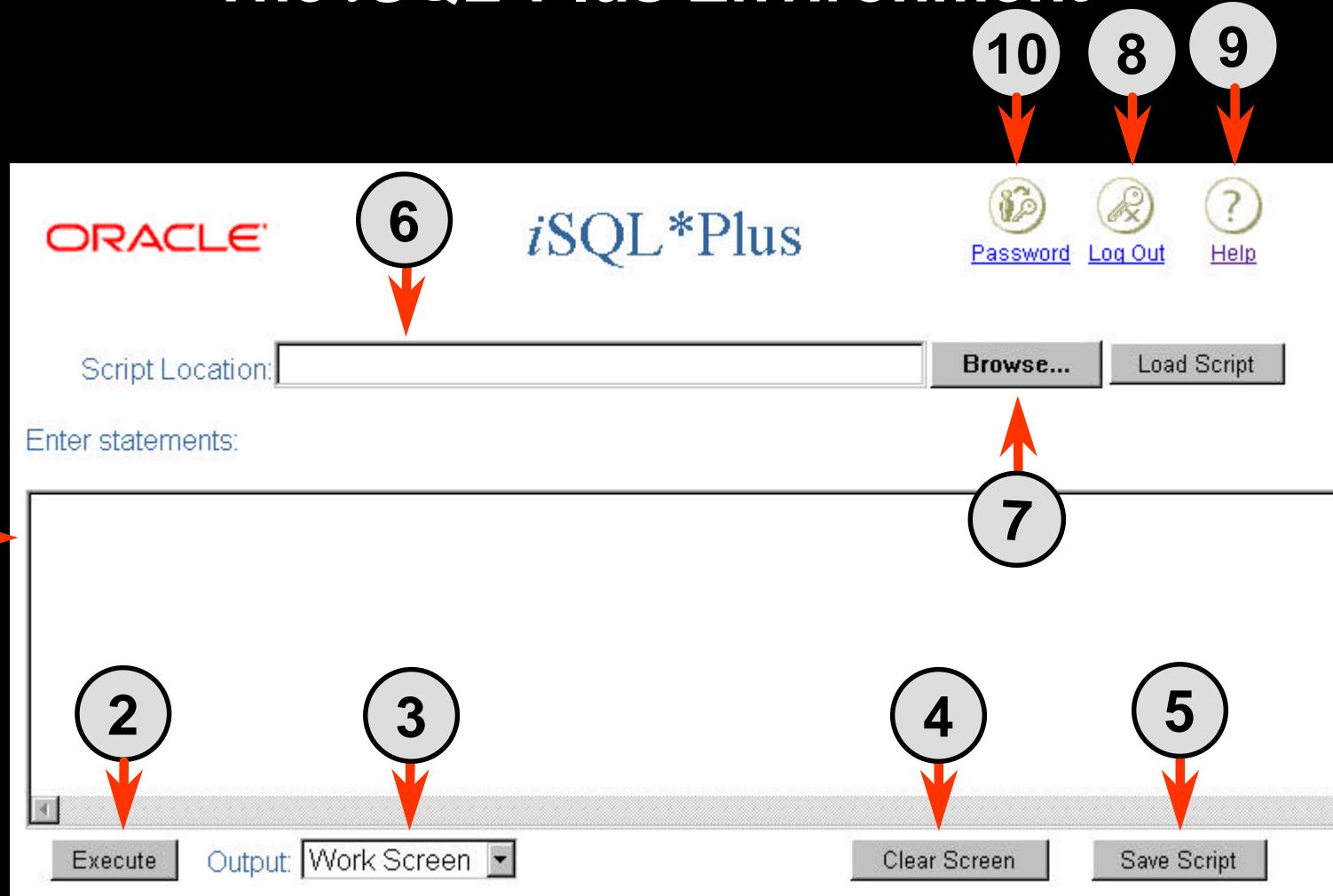
- **Describe the table structure**
- **Edit your SQL statement**
- **Execute SQL from *i*SQL*Plus**
- **Save SQL statements to files and append SQL statements to files**
- **Execute statements stored in saved files**
- **Load commands from a text file into the *i*SQL*Plus Edit window**

Logging In to *iSQL*Plus*

From your Windows browser environment:



The *i*SQL*Plus Environment



Displaying Table Structure

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

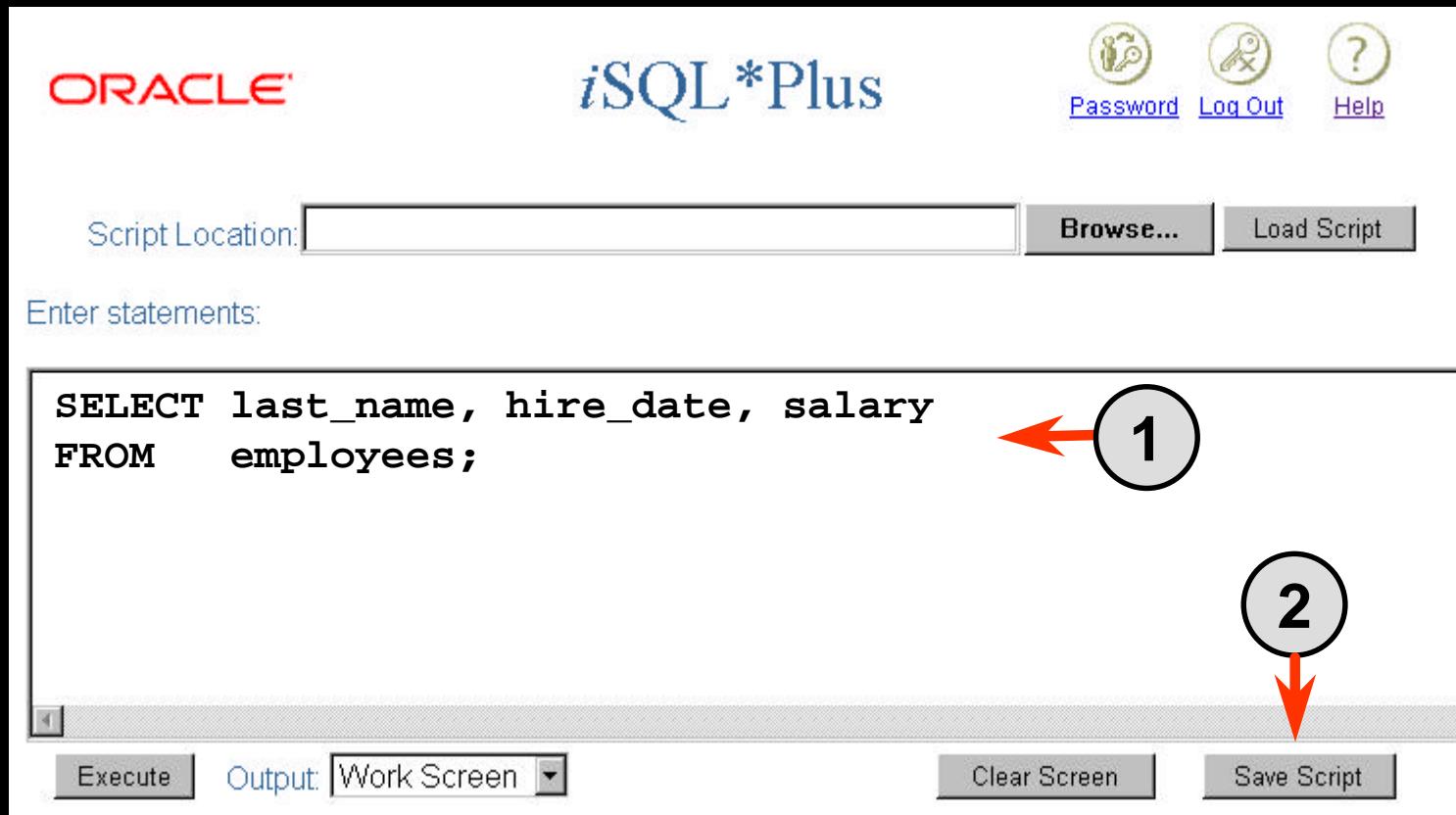
```
DESC[RIBE] tablename
```

Displaying Table Structure

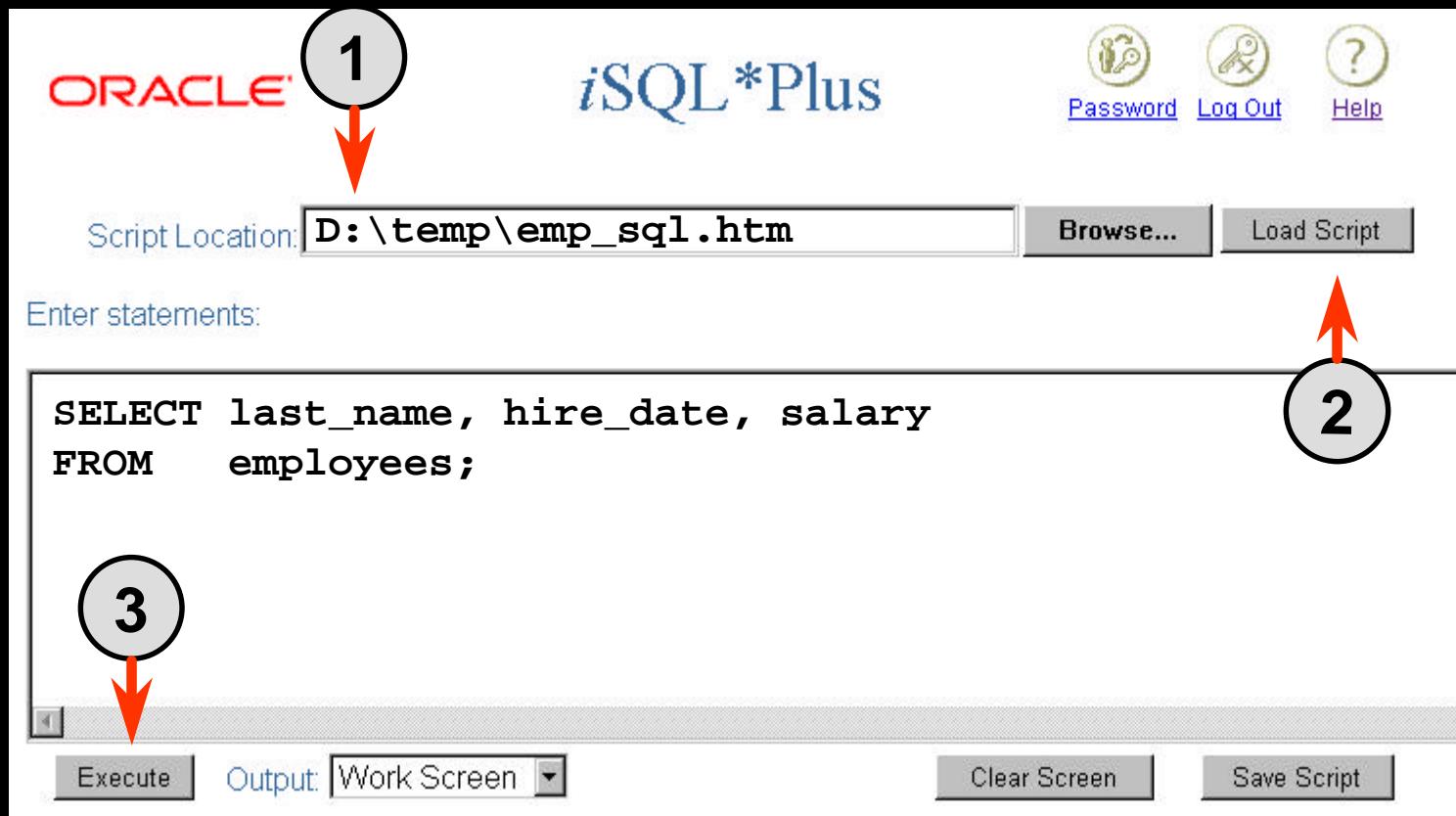
DESCRIBE employees

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)

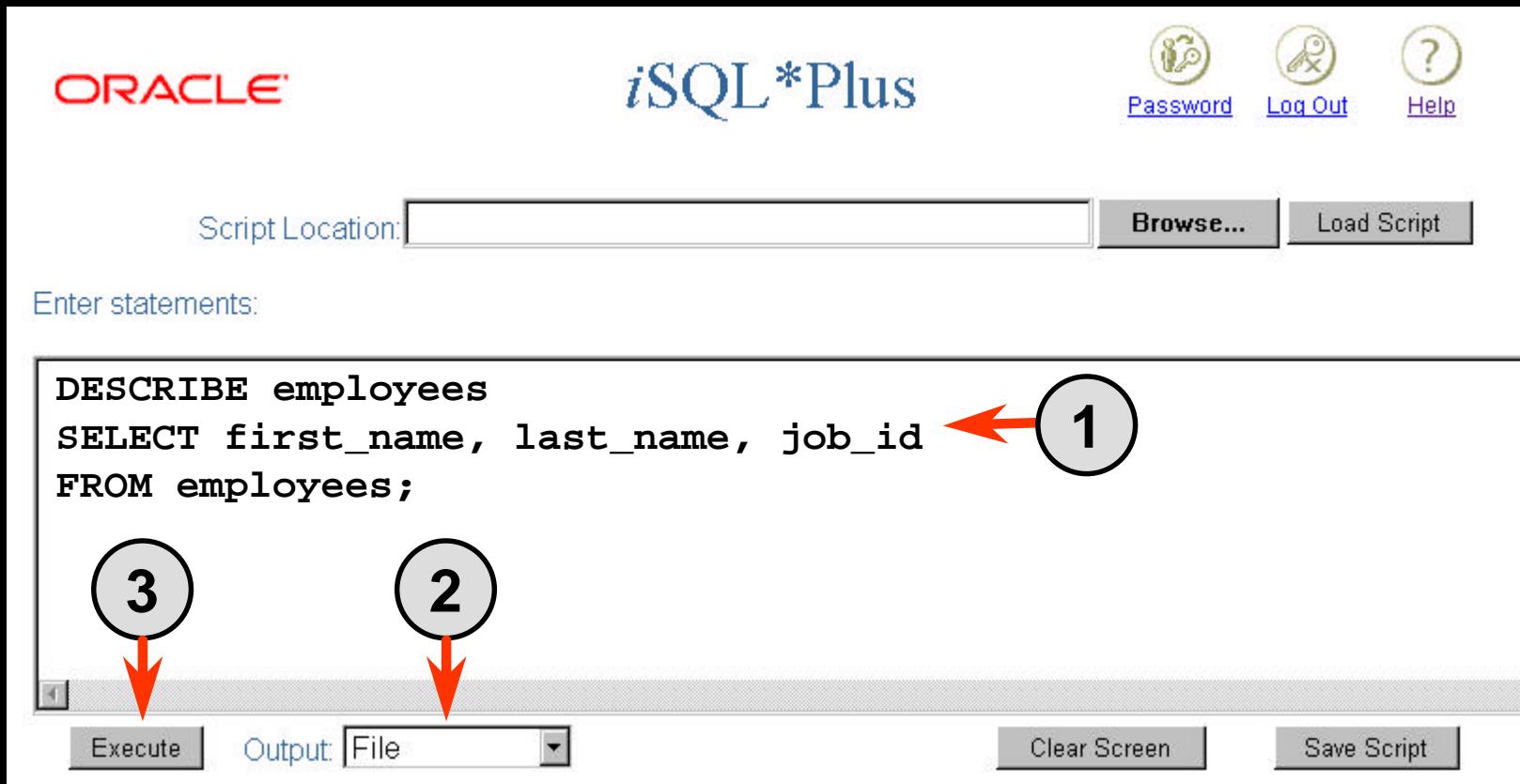
Interacting with Script Files



Interacting with Script Files



Interacting with Script Files



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 give descriptive column headings
- Use the *iSQL*Plus* environment to write, save, and execute SQL statements and *iSQL*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

ORACLE®

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Limit the rows retrieved by a query**
- **Sort the rows retrieved by a query**

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	Mouraos	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 Selected

- Restrict the rows 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 in 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 = 'Goyal';
```

Comparison Conditions

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

Using Comparison Conditions

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

LAST_NAME	SALARY
Matos	2600
Vargas	2500

Other Comparison Conditions

Operator	Meaning
BETWEEN . . . AND . . .	Between two values (inclusive)
IN (set)	Match any of a list of values
LIKE	Match a character pattern
IS NULL	Is a null value

Using 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 <i>both</i> component conditions are true
OR	Returns TRUE if <i>either</i> 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.



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.



Rules of Precedence

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

Override rules of precedence by using parentheses.

Rules of Precedence

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

LAST_NAME	JOB_ID	SALARY
King	AD_PRES	24000
Abel	SA_REP	11000
Taylor	SA_REP	8600
Grant	SA_REP	7000

Rules of Precedence

Use parentheses to force priority.

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

LAST_NAME	JOB_ID	SALARY
King	AD_PRES	24000

ORDER BY Clause

- Sort rows with the ORDER BY clause
 - ASC: ascending order (the default order)
 - 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
Randall	AD_VP	90	13-JAN-92

20 rows selected.



Sorting in Descending Order

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

LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
Zlotkey	SA_MAN	80	29-JAN-00
Mourgos	ST_MAN	50	16-NOV-99
Grant	SA_REP		24-MAY-99
Lorentz	IT_PROG	60	07-FEB-99
Vargas	ST_CLERK	50	09-JUL-98
Taylor	SA_REP	80	24-MAR-98
Matos	ST_CLERK	50	15-MAR-98
Fay	MK_REP	20	17-AUG-97
Davies	ST_CLERK	50	29-JAN-97
Abel	SA_REP	80	11-MAY-96

King	AD_PRES	90	17-JUN-87
------	---------	----	-----------

20 rows selected.



Sorting by Column Alias

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

EMPLOYEE_ID	LAST_NAME	ANNSAL
144	Vargas	30000
143	Matos	31200
142	Davies	37200
141	Rajs	42000
107	Lorentz	50400
200	Whalen	52800
124	Mourgos	69600
104	Ernst	72000
202	Fay	72000
178	Grant	84000
206	Gietz	99600

20 rows selected.

Sorting by Multiple Columns

- The order of ORDER BY list is the order of sort.

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

LAST_NAME	DEPARTMENT_ID	SALARY
Whalen	10	4400
Hartstein	20	13000
Fay	20	6000
Mourgos	50	5800
Rajs	50	3500

Higgins	110	12000
Gietz	110	8300
Grant		7000

20 rows selected.

- You can sort by a column that is not in the SELECT list.



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]];
```

Practice 2 Overview

This practice covers the following topics:

- **Selecting data and changing the order of rows displayed**
- **Restricting rows by using the WHERE clause**
- **Sorting rows by using the ORDER BY clause**

3

Single-Row Functions

ORACLE®

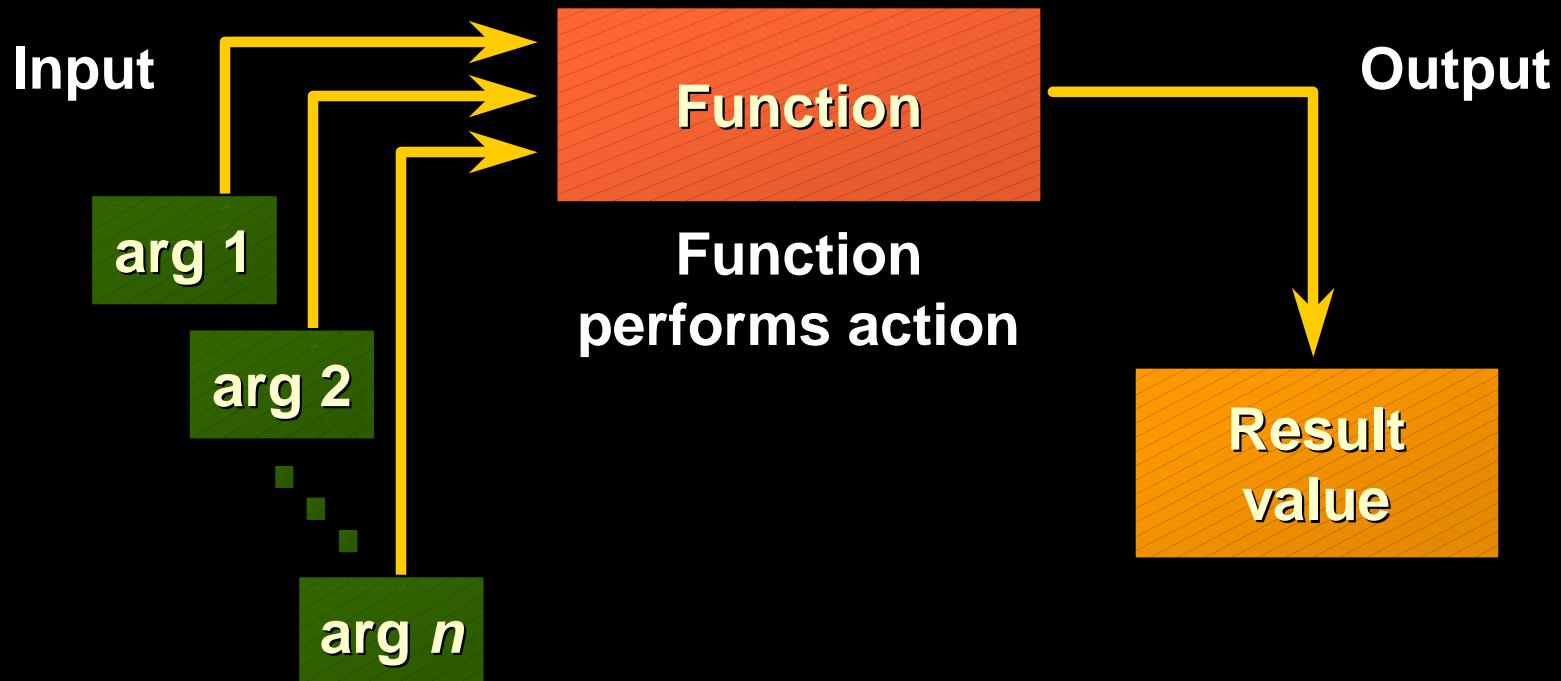
Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

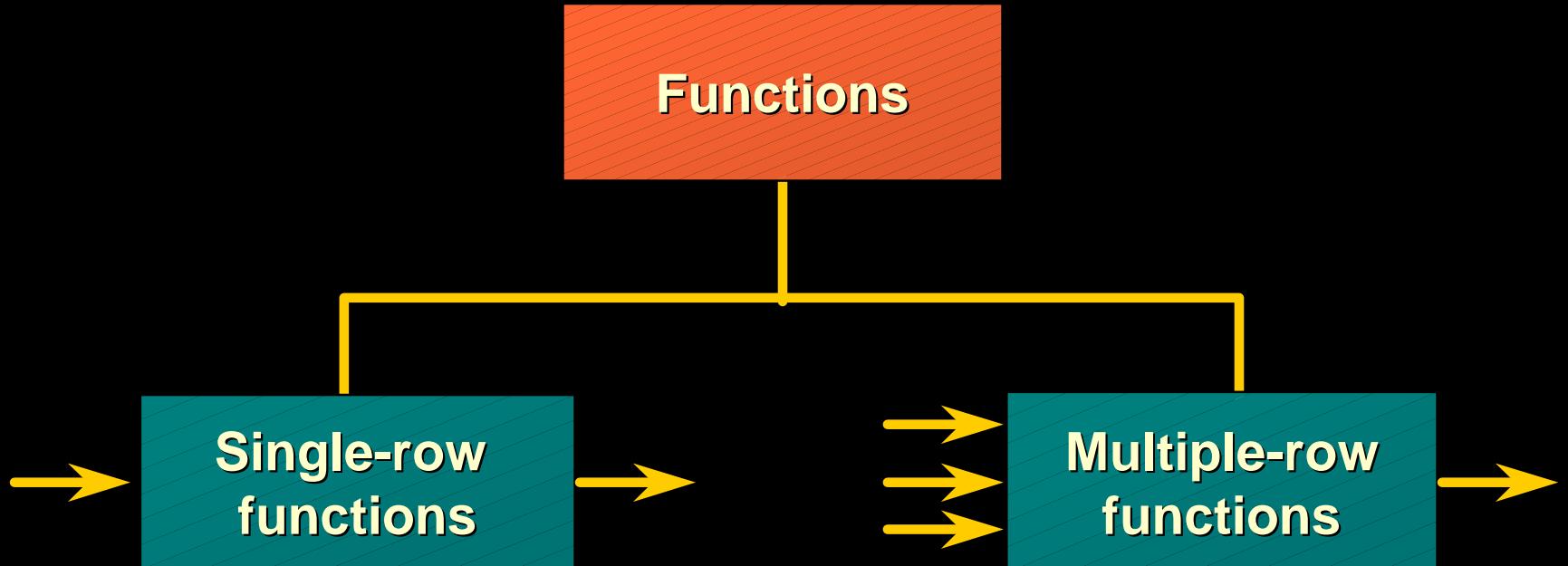
After completing this lesson, you should be able to do the following:

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

SQL Functions



Two Types of SQL Functions



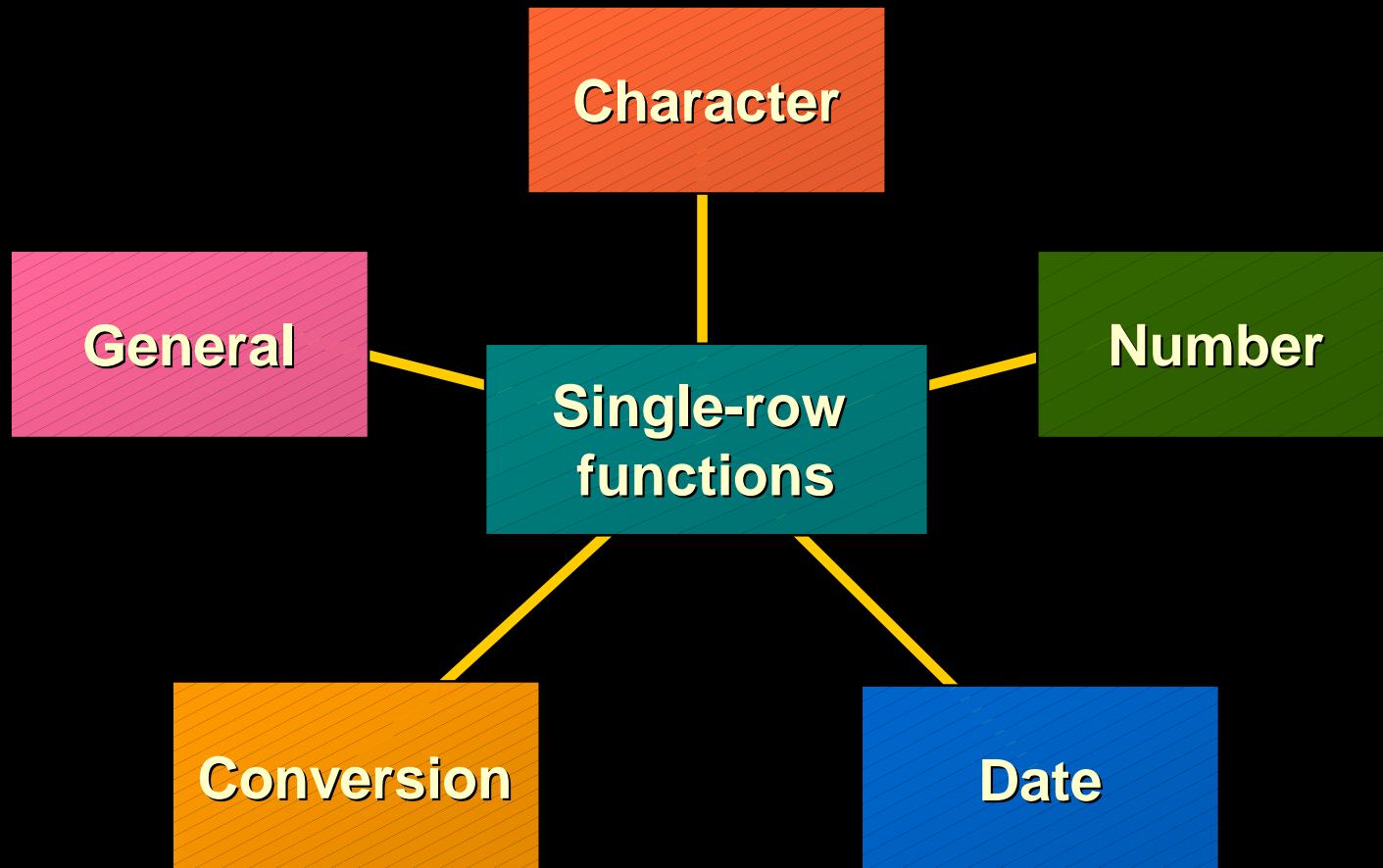
Single-Row Functions

Single row functions:

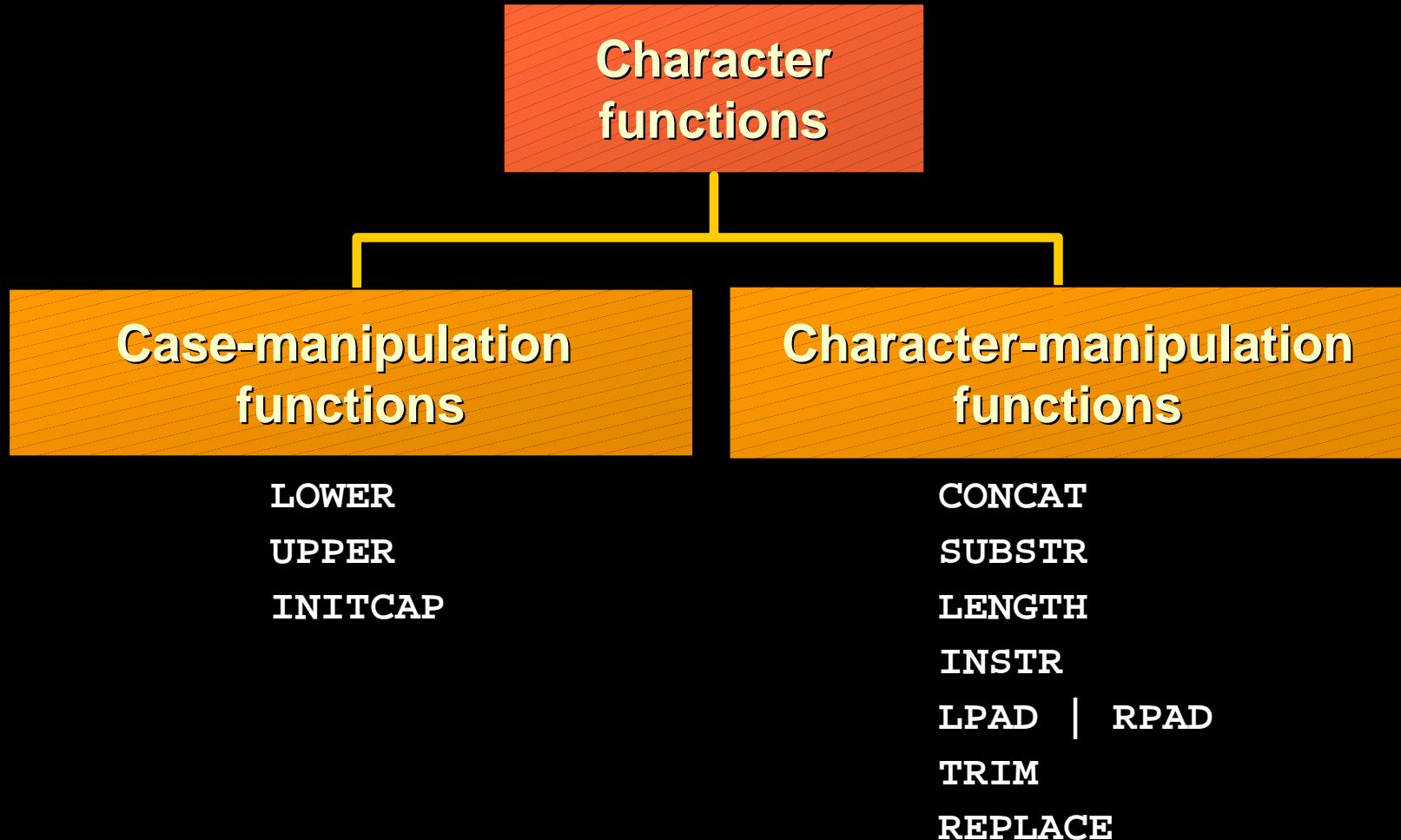
- Manipulate data items
- Accept arguments and return one value
- Act on each row returned
- Return one result per row
- May modify the data type
- Can be nested
- Accept arguments which can be a column or an expression

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

Single-Row Functions



Character Functions



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
<code>CONCAT('Hello', 'World')</code>	<code>HelloWorld</code>
<code>SUBSTR('HelloWorld',1,5)</code>	<code>Hello</code>
<code>LENGTH('HelloWorld')</code>	<code>10</code>
<code>INSTR('HelloWorld', 'W')</code>	<code>6</code>
<code>LPAD(salary,10,'*')</code>	<code>*****24000</code>
<code>RPAD(salary, 10, '*')</code>	<code>24000*****</code>
<code>TRIM('H' FROM 'HelloWorld')</code>	<code>elloWorld</code>

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

`ROUND(45.926, 2)`  45.93

- **TRUNC:** Truncates value to specified decimal

`TRUNC(45.926, 2)`  45.92

- **MOD:** Returns remainder of division

`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 you can use to view results from functions and calculations.

Using the TRUNC Function

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

TRUNC(45.923,2)	TRUNC(45.923)	TRUNC(45.923,-2)
45.92	45	0

Using the MOD Function

Calculate the remainder of a salary after it is divided by 5000 for all employees whose job title is sales representative.

```
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

Working with Dates

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

```
SELECT last_name, hire_date  
FROM   employees  
WHERE  last_name like 'G%';
```

LAST_NAME	HIRE_DATE
Gietz	07-JUN-94
Grant	24-MAY-99



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	716.227563
Kochhar	598.084706
De Haan	425.227563

Date Functions

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

Using Date Functions

- `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'**

Using Date Functions

Assume SYSDATE = '25-JUL-95':

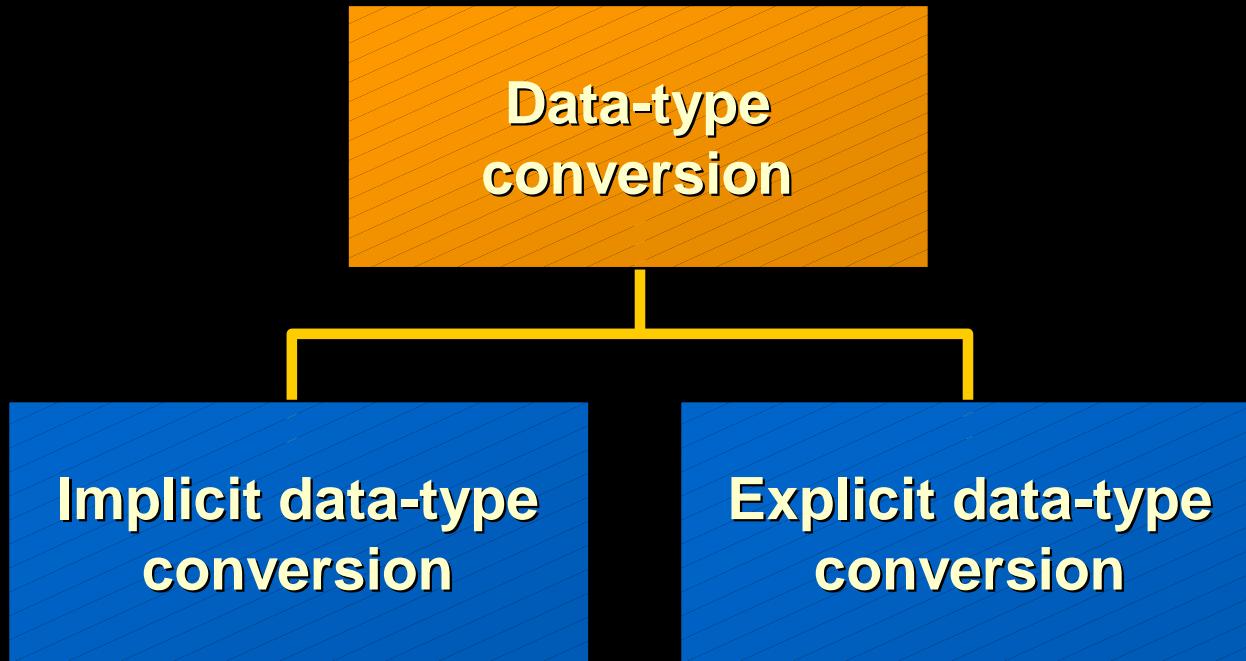
- **ROUND(SYSDATE, 'MONTH')** → 01-AUG-95
- **ROUND(SYSDATE, 'YEAR')** → 01-JAN-96
- **TRUNC(SYSDATE, 'MONTH')** → 01-JUL-95
- **TRUNC(SYSDATE, 'YEAR')** → 01-JAN-95

Practice 3, Part 1 Overview

This practice covers the following topics:

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

Conversion Functions



Implicit Data-Type Conversion

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

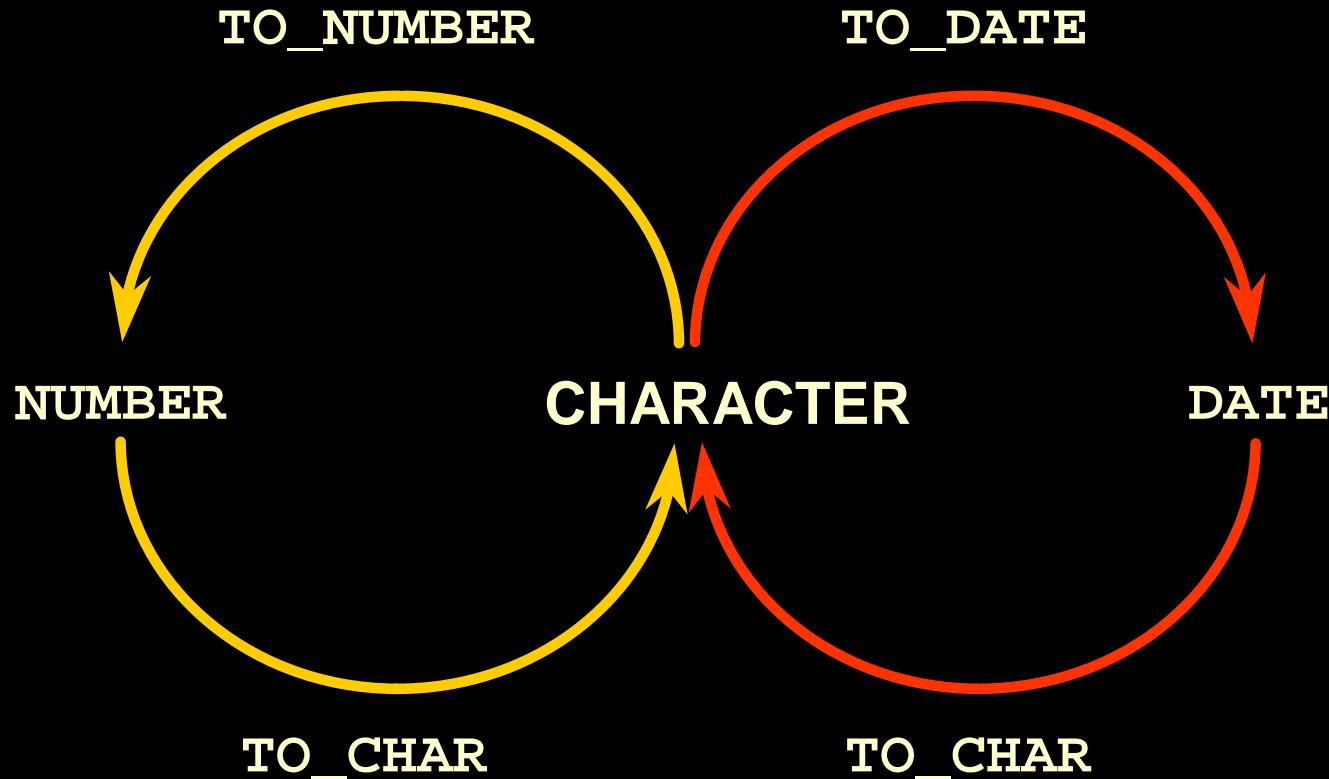
From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE
NUMBER	VARCHAR2
DATE	VARCHAR2

Implicit Data-Type Conversion

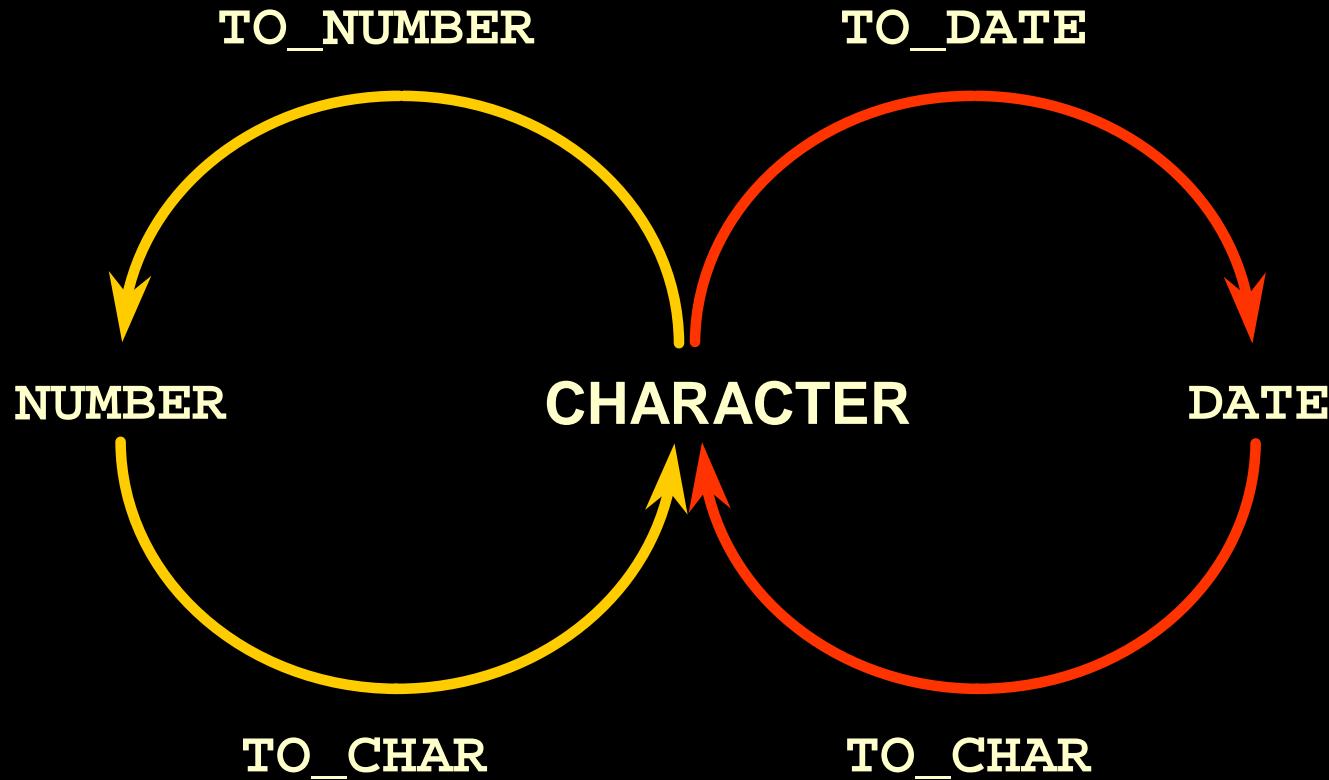
For expression evaluation, the Oracle Server can automatically convert the following:

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 in single quotation marks and **is case sensitive**
- Can include any valid date format element
- Has an *fm* element to remove padded blanks or suppress leading zeros
- Is separated from the date value by a comma

Elements of the Date Format Model

YYYY	Full year in numbers
YEAR	Year spelled out
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') 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
Rais	17 October 1995
Gietz	7 June 1994

20 rows selected.



Using the TO_CHAR Function with Numbers

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

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

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 thousand indicator

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

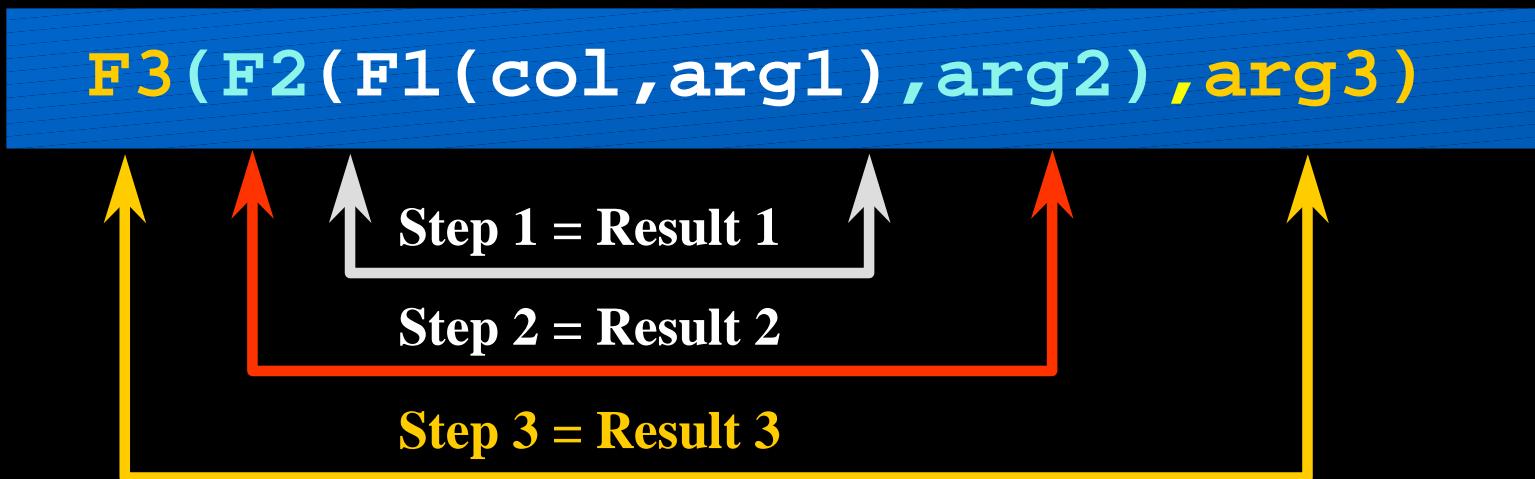
To find employees hired prior to 1990, use the RR format, which produces the same results whether the command is run in 1999 or now:

```
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.



Nesting Functions

```
SELECT last_name,  
       NVL(TO_CHAR(manager_id), 'No Manager')  
FROM   employees  
WHERE  manager_id IS NULL;
```

LAST_NAME	NVL(TO_CHAR(MANAGER_ID),'NOMANAGER')
King	No Manager

General Functions

These functions work with any data type and pertain to using null value.

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

NVL Function

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

Using the NVL Function

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

LAST_NAME	SALARY	NVL(COMMISSION_PCT,0)	AN_SAL
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
Davies	3100	0	37200
Matos	2600	0	31200
Vargas	2500	0	30000
Zlotkey	10500	.2	151200
Abel	11000	.3	171600

20 rows selected.



Using the NVL2 Function

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

LAST_NAME	SALARY	COMMISSION_PCT	INCOME
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.

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
William	7	Gietz	5	7
Shelley	7	Higgins	7	
Pat	3	Fay	3	
Michael	7	Hartstein	9	7
Jennifer	8	Whalen	6	8
Kimberely	9	Grant	5	9
Jonathon	8	Taylor	6	8
Ellen	5	Abel	4	5
Eleni	5	Zlotkey	7	5
Peter	5	Vargas	6	5
Randall	7	Matos	5	7
Curtis	6	Davies	6	
Trenna	6	Rajs	4	6
Karen	5	Mikkilineni	7	5

20 rows selected.



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, it returns that expression; otherwise, it does a COALESCE of the remaining expressions.

Using the COALESCE Function

```
SELECT      last_name,  
            COALESCE(commission_pct, salary, 10) comm  
FROM        employees  
ORDER BY    commission_pct;
```

LAST_NAME	COMM
Grant	.15
Zlotkey	.2
Taylor	.2
Abel	.3
King	24000
Kochhar	17000
De Haan	17000
Hunold	9000

Matos	2600
Vargas	2500

20 rows selected.



Conditional Expressions

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

The 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;
```

Lorentz	IT_PROG	4200	4620
Mourgos	ST_MAN	5800	5800
Rajs	ST_CLERK	3500	4025

Gietz	AC_MGR	8300	8300
	AC_ACCOUNT	8300	8300

20 rows selected.



The DECODE Function

Facilitates conditional inquiries by doing the work of a CASE or 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;
```

Lorentz	IT_PROG	4200	4620
Mourgos	ST_MAN	5800	5800
Rajs	ST_CLERK	3500	4025

Gietz	AC_ACCOUNT	8300	8300
-------	------------	------	------

20 rows selected.

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, Part 2 Overview

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

Displaying Data from Multiple Tables

ORACLE®

Copyright © Oracle Corporation, 2001. All rights reserved.

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 equality and nonequality joins**
- **View data that generally does not meet a join condition by using outer joins**
- **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
205	Higgins	110
206	Gietz	110

20 rows selected.

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
190	Contracting	1700

8 rows selected.

EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
200	10	Administration
201	20	Marketing
202	20	Marketing
124	50	Shipping
141	50	Shipping

205	110	Accounting
206	110	Accounting

19 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 in a WHERE clause.

Generating a Cartesian Product

EMPLOYEES (20 rows)

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90

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

190	Contracting	1700
-----	-------------	------

8 rows selected.

Cartesian
product: →

$20 \times 8 = 160$ rows

EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
200	10	Administration
201	20	Marketing
202	20	Marketing
124	50	Shipping
141	50	Shipping

206	110	Contracting
-----	-----	-------------

160 rows selected.

Types of Joins

Oracle Proprietary Joins (*8i* and prior):

- **Equijoin**
- **Nonequijoin**
- **Outer join**
- **Self join**

SQL: 1999 Compliant Joins:

- **Cross joins**
- **Natural joins**
- **Using clause**
- **Full or two sided outer joins**
- **Arbitrary join conditions for outer joins**

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.

What Is an Equijoin?

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
205	110
206	110

19 rows selected.

DEPARTMENTS

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



Foreign key



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
205	Higgins	110	110	1700
206	Gietz	110	110	1700

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
Lorentz	60
Zlotkey	80

Gietz	110
-------	-----

19 rows selected.

DEPARTMENTS

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

Accounting	110
------------	-----

Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Improve performance by using table prefixes.
- Distinguish columns that have identical names but reside in different tables by using column aliases.

Using Table Aliases

- Simplify queries by using table aliases
- Improve performance by using table prefixes

```
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
Grant	
Whalen	10
Hartstein	20
Fay	20
Higgins	110
Gietz	110

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.

Nonequi joins

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

Fay	15000
Higgins	12000
Gietz	8300

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

6 rows selected.



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

Retrieving Records with NonequiJoins

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

LAST_NAME	SALARY	GRA
Matos	2600	A
Vargas	2500	A
Lorentz	4200	B
Mourgos	5800	B
Rajs	3500	B
Davies	3100	B
Kochhar	17000	E
De Haan	17000	E

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

10	Willman
20	Hartstein
20	Fay
110	Higgins
110	Gietz

20 rows selected.



There are no employees in department 190.

Outer Joins Syntax

- You use an outer join to also 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

Higgins	110	Accounting
Gietz	110	Accounting
		Contracting

20 rows selected.

Self Joins

EMPLOYEES (WORKER)

EMPLOYEE_ID	LAST_NAME	MANAGER_ID
101	Kochhar	100
102	De Haan	100
124	Mourgos	100
149	Zlotkey	100
201	Hartstein	100
200	Whalen	101

EMPLOYEES (MANAGER)

EMPLOYEE_ID	LAST_NAME
100	King
101	Kochhar

19 rows selected.

206 Gietz

205 Higgins



**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;
```

W.LAST_NAME 'WORKSFOR' M.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
Fay works for Hartstein
Gietz works for Higgins

19 rows selected.

Practice 4, Part 1 Overview

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



Joining Tables Using SQL: 1999 Syntax

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

```
SELECT      table1.column, table2.column
FROM        table1
[CROSS JOIN table2] |
[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)];
```

Creating Cross Joins

- The **CROSS JOIN clause produces the cross-product of two tables.**
- This is the same as a **Cartesian product between the two tables.**

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

LAST_NAME	DEPARTMENT_NAME
Koch	Marketing
Fay	Contracting
Higgins	Contracting
Gietz	Contracting

160 rows selected.

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, then 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.

Note: 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.

Retrieving Records with the USING Clause

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

	EMPLOYEE_ID	LAST_NAME	LOCATION_ID
	200	Whalen	1700
	201	Hartstein	1800
	202	Fay	1800
	124	Mourgos	1500
	141	Rajs	1500
	142	Davies	1500
	205	Higgins	1700
	206	Gietz	1700

19 rows selected.

Creating Joins with the ON Clause

- The join condition for the natural join is basically an equijoin of all columns with the same name.
- To specify arbitrary conditions or specify columns to join, the ON clause is used.
- Separates the join condition 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
205	Higgins	110	110	1700
206	Gietz	110	110	1700

19 rows selected.



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
100	Seattle	Executive
101	Seattle	Executive
102	Seattle	Executive
103	Southlake	IT
104	Southlake	IT
107	Southlake	IT
124	South San Francisco	Shipping



19 rows selected.

INNER versus OUTER Joins

- In SQL: 1999, the join of two tables returning only matched rows is an inner join.
- A join between two tables that returns the results of the inner join as well as unmatched rows left (or right) tables is 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
	King	90	Executive
	Kochhar	90	Executive

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Ernst	60	IT
Grant		
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Higgins	110	Accounting
Gietz	110	Accounting

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
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.

FULL OUTER JOIN

```
SELECT e.last_name, e.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
Abel	80	Sales
Davies	50	Shipping
De Haan	90	Executive
Ernst	60	IT
Fay	20	Marketing
Gietz	110	Accounting
Grant		
Hartstein	20	Marketing

Zlotkey	80	Sales
		Contracting

21 rows selected.

Additional Conditions

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
  FROM employees e JOIN departments d
 WHERE (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

Summary

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

- Oracle proprietary syntax for versions 8*i* and earlier
- SQL: 1999 compliant syntax for version 9*i*

Practice 4, Part 2 Overview

This practice covers the following topics:

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



Aggregating Data Using 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 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

20	5000
110	12000
110	8300

20 rows selected.

The maximum salary in the 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 any data type.

```
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.

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

COUNT(*)
5

Using the COUNT Function

- COUNT(*expr*) returns the number of rows with non-null values for the *expr*.
- Display the number of department values in the EMPLOYEES table, excluding the null values.

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

COUNT(COMMISSION_PCT)
3

Using the DISTINCT Keyword

- COUNT(DISTINCT *expr*) returns the number of distinct **nonnull values** of the *expr*.
- 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.

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

AVG(COMMISSION_PCT)
.2125

Using the NVL Function with Group Functions

The NVL function forces group functions to include null values.

```
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
110	8300
	7000

20 rows selected.

4400
9500
3500
6400
The average salary in EMPLOYEES table for each department.

DEPARTMENT_ID	AVG(SALARY)
10	4400
20	9500
50	3500
60	6400
80	10922.2222

8 rows selected.

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] ;
```

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

8 rows selected.



Grouping by More Than One Column

EMPLOYEES

DEPARTMENT_ID	JOB_ID	SALARY
10	AD_ASST	4400
20	MK_MAN	13000
20	MK_REP	6000
50	ST_CLERK	3500
50	ST_CLERK	3100
50	ST_CLERK	2600
50	ST_CLERK	2500
50	ST_MAN	5800
60	IT_PROG	9000
60	IT_PROG	6000
60	IT_PROG	4200
80	SA_MAN	10500
80	SA_REP	11000
110	AC_MGR	12000
	SA_REP	7000

20 rows selected.

Add up 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	24000

|SA_REP|

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;
```

Column missing in the GROUP BY clause

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

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

Excluding Group Results

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



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

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

Excluding Group Results: The HAVING Clause

Use the HAVING clause to restrict groups:

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

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

Using the HAVING Clause

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

DEPARTMENT_ID	MAX(SALARY)
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, AVG
- Write queries that use the GROUP BY clause
- Write queries that use the HAVING clause

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

Practice 5 Overview

This practice covers the following topics:

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

Subqueries

Objectives

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

- **Describe the types of problem that subqueries can solve**
- **Define subqueries**
- **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.
- The result of the subquery is used by the main query (outer query).

Using a Subquery

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

LAST_NAME
King
Kochhar
De Haan
Hartstein
Higgins

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

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);
```

LAST_NAME	JOB_ID	SALARY
Vargas	ST_CLERK	2500

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) > 2500
            (SELECT MIN(salary)
             FROM employees
              WHERE department_id = 50);
```

What Is Wrong with This Statement?

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

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

Single-row operator with
multiple-row subquery

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

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

206 Gietz AC_ACCOUNT 6300

10 rows selected.

Using the ALL Operator in Multiple-Row Subqueries

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

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

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**



Producing Readable Output with *i*SQL*Plus

ORACLE®

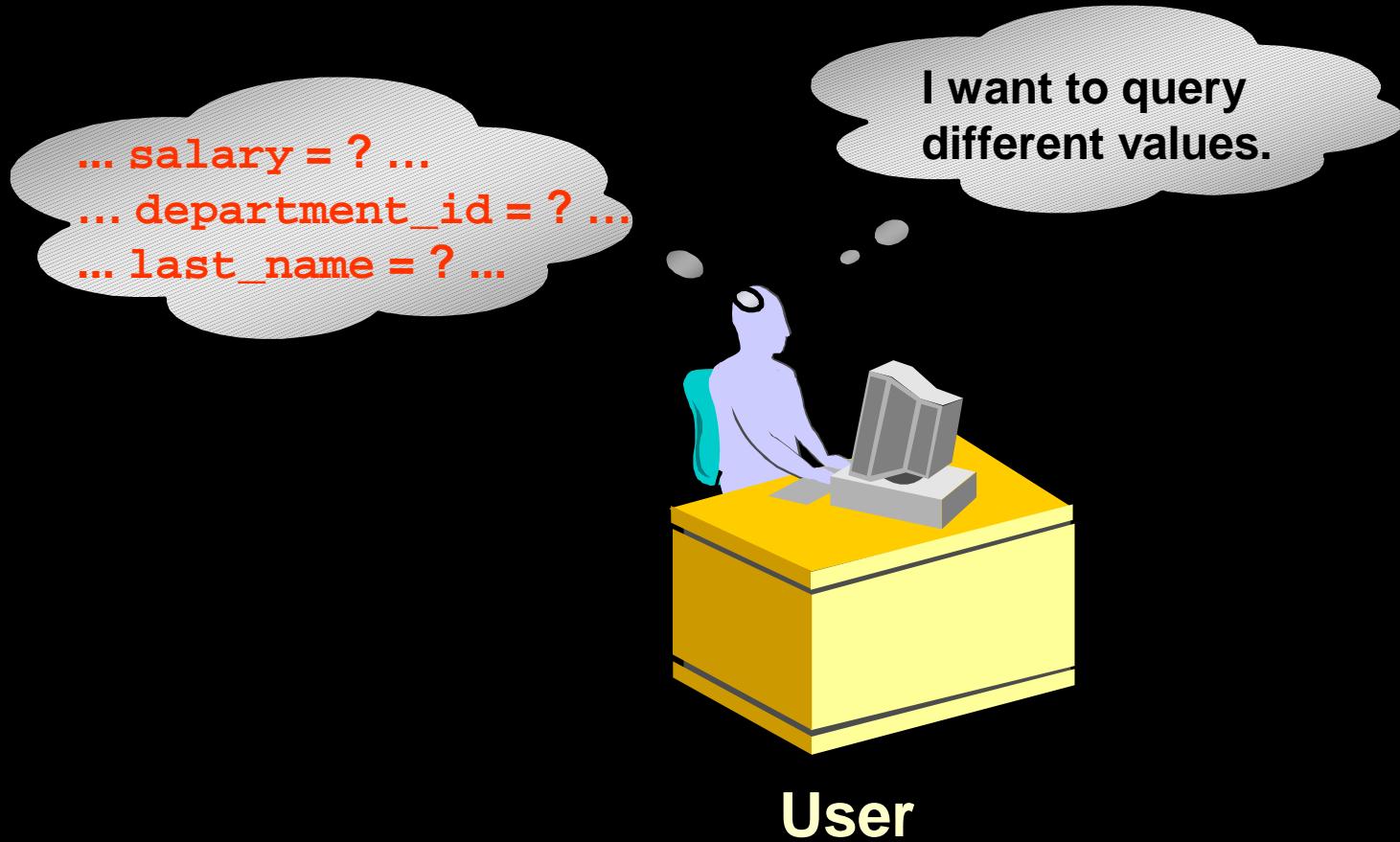
Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- Produce queries that require a substitution variable
- Customize the *iSQL*Plus* environment
- Produce more readable output
- Create and execute script files

Substitution Variables



User

Substitution Variables

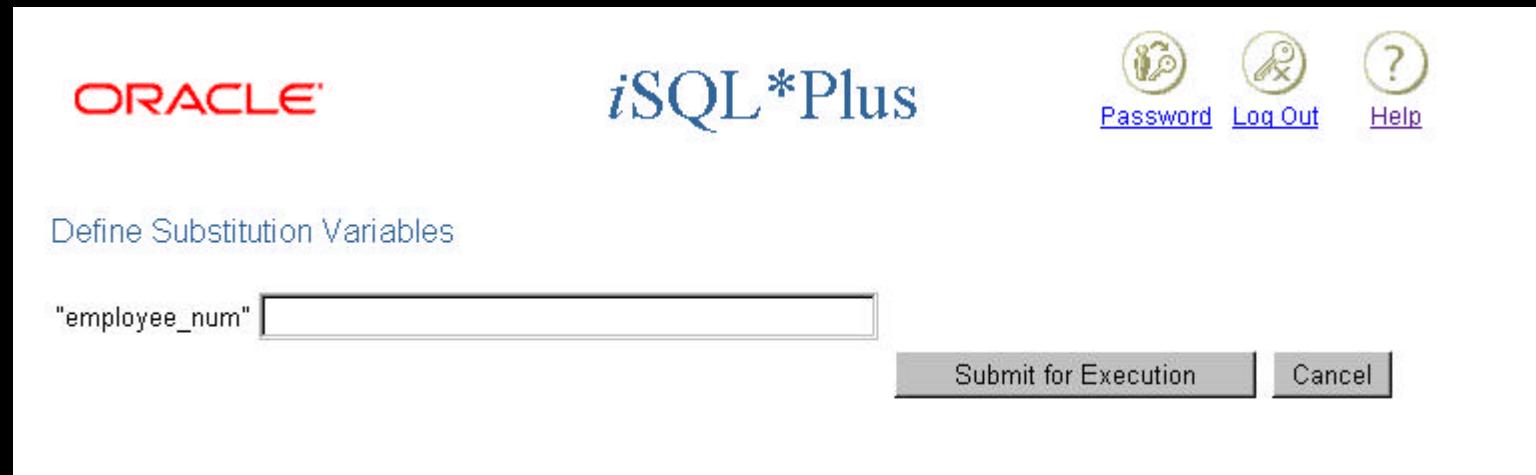
Use iSQL*Plus substitution variables to:

- **Store values temporarily**
 - Single ampersand (&)
 - Double ampersand (&&)
 - **DEFINE command**
- **Pass variable values between SQL statements**
- **Dynamically alter headers and footers**

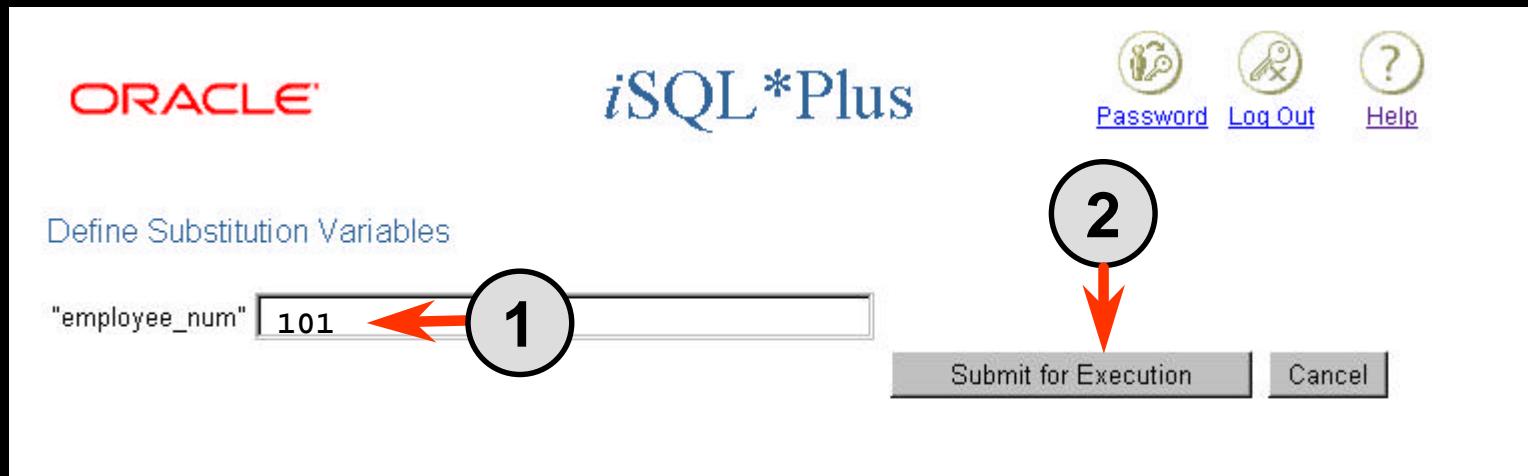
Using the & Substitution Variable

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';
```

Define Substitution Variables

"job_title"

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



Specifying Column Names, Expressions, and Text

Use substitution variables to supplement the following:

- **WHERE conditions**
- **ORDER BY clauses**
- **Column expressions**
- **Table names**
- **Entire SELECT statements**

Specifying Column Names, Expressions, and Text

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

"column_name"

"condition"

"order_column"

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
102	De Haan	AD_VP	17000
100	King	AD_PRES	24000
101	Kochhar	AD_VP	17000

Defining Substitution Variables

- You can **predefine variables using the *i*SQL*Plus DEFINE command.**
`DEFINE variable = value` creates a user variable with the CHAR data type.
- If you need to **predefine a variable that includes spaces, you must enclose the value within single quotation marks when using the DEFINE command.**
- A **defined variable is available for the session**

DEFINE and UNDEFINE Commands

- A variable remains defined until you either:
 - Use the UNDEFINE command to clear it
 - Exit *iSQL*Plus*
- You can verify your changes with the DEFINE command.

```
DEFINE job_title = IT_PROG
DEFINE job_title
DEFINE JOB_TITLE          = "IT_PROG" (CHAR)
```

```
UNDEFINE job_title
DEFINE job_title
SP2-0135: symbol job_title is UNDEFINED
```

Using the DEFINE Command with & Substitution Variable

- Create the substitution variable using the DEFINE command.

```
DEFINE employee_num = 200
```

- Use a variable prefixed with an ampersand (&) to substitute the value in the SQL statement.

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

EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID
200	Whalen	4400	10

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;
```

"column_name"

EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
200	Whalen	AD_ASST	10
201	Hartstein	MK_MAN	20
202	Fay	MK_REP	20
114	Raphealy	AC_MGR	30
124	Mourgos	ST_MAN	50
141	Rajs	ST_CLERK	50

Using the VERIFY Command

Use the VERIFY command to toggle the display of the substitution variable, before and after iSQL*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
```



Customizing the iSQL*Plus Environment

- Use SET commands to control current session.

```
SET system_variable value
```

- Verify what you have set by using the SHOW command.

```
SET ECHO ON
```

```
SHOW ECHO  
echo ON
```

SET Command Variables

- **ARRAYSIZE** {20 | *n*}
- **FEEDBACK** {6 | *n* | OFF | ON}
- **HEADING** {OFF | ON}
- **LONG** {80 | *n*} | ON | *text*}

```
SET HEADING OFF
```

```
SHOW HEADING  
HEADING OFF
```

*i*SQL*Plus Format Commands

- **COLUMN** [*column option*]
- **TTITLE** [*text* | OFF | ON]
- **BTITLE** [*text* | OFF | ON]
- **BREAK** [ON *report_element*]

The COLUMN Command

Controls display of a column:

```
COL[UMN] [{column|alias} [option]]
```

- **CLE[AR]: Clears any column formats**
- **HEA[DING] *text*: Sets the column heading**
- **FOR[MAT] *format*: Changes the display of the column using a format model**
- **NOPRINT | PRINT**
- **NULL**

Using the COLUMN Command

- Create column headings.

```
COLUMN last_name HEADING 'Employee|Name'  
COLUMN salary JUSTIFY LEFT FORMAT $99,990.00  
COLUMN manager FORMAT 999999999 NULL 'No manager'
```

- Display the current setting for the LAST_NAME column.

```
COLUMN last_name
```

- Clear settings for the LAST_NAME column.

```
COLUMN last_name CLEAR
```

COLUMN Format Models

Element	Description	Example	Result
9	Single zero-suppression digit	999999	1234
0	Enforces leading zero	099999	001234
\$	Floating dollar sign	\$9999	\$1234
L	Local currency	L9999	L1234
.	Position of decimal point	9999.99	1234.00
,	Thousand separator	9,999	1,234

Using the BREAK Command

Use the BREAK command to suppress duplicates

```
BREAK ON job_id
```

Using the TTITLE and BTITLE Commands

- Display headers and footers.

```
TTITLE [TLE] [text | OFF | ON]
```

- Set the report header.

```
TTITLE 'Salary Report'
```

- Set the report footer.

```
BTITLE 'Confidential'
```

Creating a Script File to Run a Report

- 1. Create and test the SQL SELECT statement.**
- 2. Save the SELECT statement into a script file.**
- 3. Load the script file into an editor.**
- 4. Add formatting commands before the SELECT statement.**
- 5. Verify that the termination character follows the SELECT statement.**

Creating a Script File to Run a Report

- 6. Clear formatting commands after the SELECT statement.**
- 7. Save the script file.**
- 8. Load the script file into the *iSQL*Plus* text window, and click the Execute button.**

Sample Report

Sat Mar 10	Employee Report	page 1
Job Category	Employee	Salary
AC_ACCOUNT	Gietz	\$8,300.00
AC_MGR	Higgins	\$12,000.00
AD_ASST	Whalen	\$4,400.00
IT_PROG	Ernst	\$6,000.00
	Hunold	\$9,000.00
	Lorentz	\$4,200.00
MK_MAN	Hartstein	\$13,000.00
MK_REP	Fay	\$6,000.00
SA_MAN	Zlotkey	\$10,500.00
SA_REP	Abel	\$11,000.00
	Grant	\$7,000.00
	Taylor	\$8,600.00
Confidential		

Summary

In this lesson, you should have learned how to:

- **Use *iSQL*Plus* substitution variables to store values temporarily**
- **Use `SET` commands to control the current *iSQL*Plus* environment**
- **Use the `COLUMN` command to control the display of a column**
- **Use the `BREAK` command to suppress duplicates and divide rows into sections**
- **Use the `TTITLE` and `BTITLE` commands to display headers and footers**

Practice 7 Overview

This practice covers the following topics:

- **Creating a query to display values using substitution variables**
- **Starting a command file containing variables**

Manipulating Data

Objectives

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

- **Describe each DML statement**
- **Insert rows into a table**
- **Update rows in a table**
- **Delete rows from a table**
- **Merge rows in 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

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

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

New
row

Insert a 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

The INSERT Statement Syntax

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

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

- Only one row is inserted at a time with this syntax.

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 within single quotation marks.

Inserting Rows with Null Values

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

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

Define Substitution Variables

"department_id"	40
"department_name"	Human Resources
"location"	2500

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	COMM
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	COMM
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	

The 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 department 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.
```

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.
```

Updating Rows: Integrity Constraint Error

```
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 number 55 does not exist

Removing a Row from a Table

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
70	Public Relations	100	1700
30	Purchasing		
50	Shipping	124	1500
60	IT	103	1400
100	Finance		
80	Sales	149	2500

Delete a row from the DEPARTMENTS table.

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
70	Public Relations	100	1700
30	Purchasing		
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500

The 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.
```



Deleting Rows: Integrity Constraint Error

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

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

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

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

```
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.



Using the WITH CHECK OPTION Keyword on DML Statements

- A subquery is used to identify the table and columns of the DML statement.
- The WITH CHECK OPTION keyword prohibits you from changing rows that are not in the subquery.

```
INSERT INTO (SELECT employee_id, last_name, email,
               hire_date, job_id, salary
                  FROM employees
                 WHERE department_id = 50 WITH CHECK OPTION)
VALUES (99998, 'Smith', 'JSMITH',
        TO_DATE('07-JUN-99', 'DD-MON-RR'),
        'ST_CLERK', 5000);
INSERT INTO
*
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

Overview of the Explicit Default Feature

- With the explicit default feature, you can use the `DEFAULT` keyword as a column value where the column default is desired.
- The addition of this feature is for compliance with the SQL: 1999 Standard.
- This allows the user to control where and when the default value should be applied to data.
- Explicit defaults can be used in `INSERT` and `UPDATE` statements.

Using Explicit Default Values

- **DEFAULT with INSERT:**

```
INSERT INTO departments
(department_id, department_name, manager_id)
VALUES (300, 'Engineering', DEFAULT);
```

- **DEFAULT with UPDATE:**

```
UPDATE departments
SET manager_id = DEFAULT WHERE department_id = 10;
```

The MERGE Statement

- Provides the ability to conditionally update or insert data into a database table
- Performs an UPDATE if the row exists and an INSERT if it is a new row:
 - Avoids separate updates
 - Increases performance and ease of use
 - Is useful in data warehousing applications

MERGE Statement Syntax

You can conditionally insert or update rows in a table by using the MERGE statement.

```
MERGE INTO table_name AS table_alias
  USING (table/view/sub_query) AS alias
  ON (join condition)
  WHEN MATCHED THEN
    UPDATE SET
      col1 = col_val1,
      col2 = col2_val
  WHEN NOT MATCHED THEN
    INSERT (column_list)
    VALUES (column_values);
```

Merging Rows

Insert or update rows in the COPY_EMP table to match the EMPLOYEES table.

```
MERGE INTO copy_emp AS c
  USING employees e
  ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
  UPDATE SET
    c.first_name      = e.first_name,
    c.last_name       = e.last_name,
    ...
    c.department_id  = e.department_id
WHEN NOT MATCHED THEN
  INSERT VALUES(e.employee_id, e.first_name, e.last_name,
                e.email, e.phone_number, e.hire_date, e.job_id,
                e.salary, e.commission_pct, e.manager_id,
                e.department_id);
```



Merging Rows

```
SELECT *
FROM COPY_EMP;
```

no rows selected

```
MERGE INTO copy_emp c
  USING employees e
  ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
    UPDATE SET
      ...
WHEN NOT MATCHED THEN
    INSERT VALUES...;
```

```
SELECT *
FROM COPY_EMP;
```

20 rows selected.



Database Transactions

A database transaction consists of one of the following:

- DML statements which constitute one consistent change to the data
- One DDL statement
- One DCL statement

Database Transactions

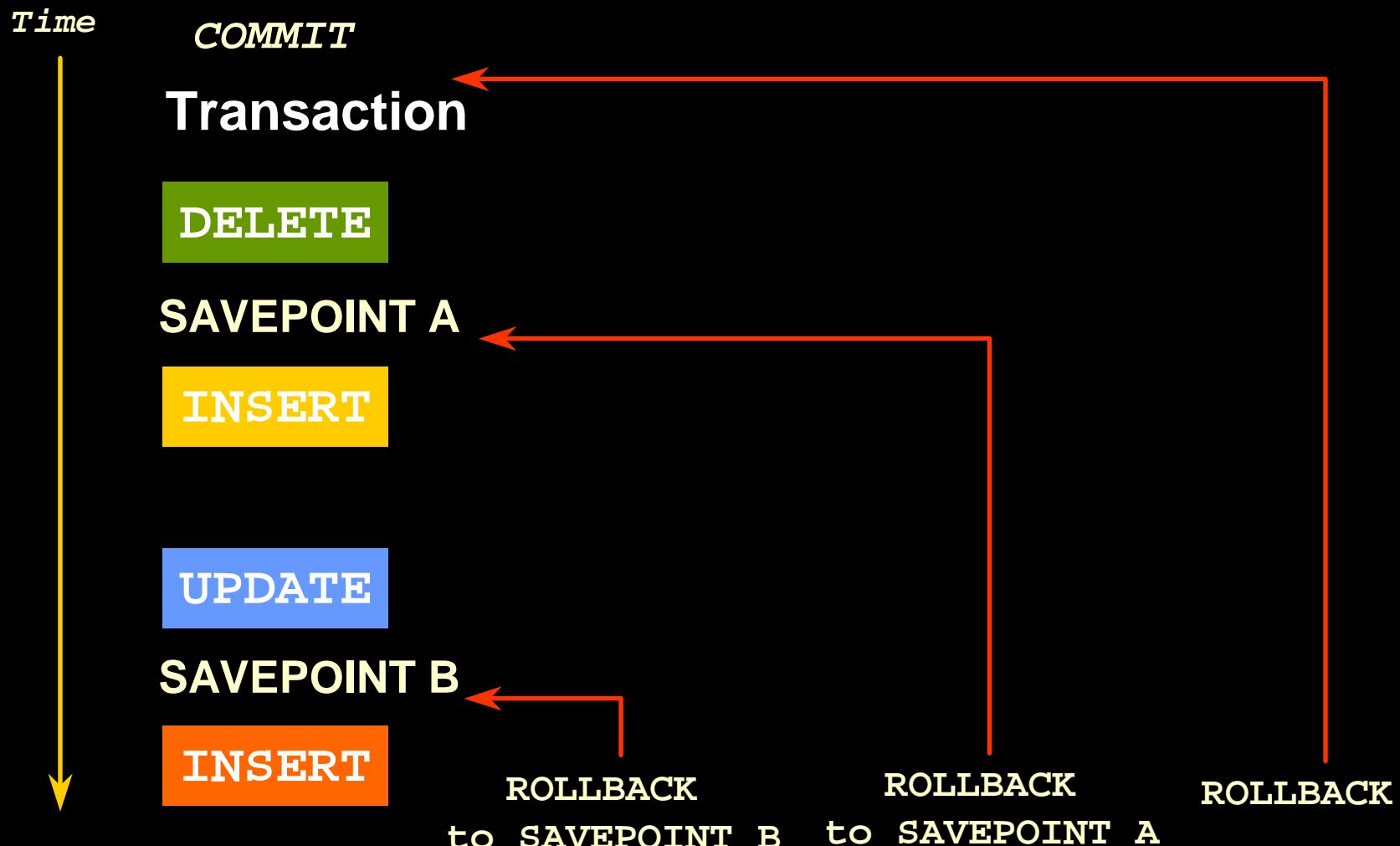
- **Begin when the first DML SQL statement is executed**
- **End with one of the following events:**
 - A COMMIT or ROLLBACK statement is issued
 - A DDL or DCL statement executes (automatic commit)
 - The user exits *iSQL*Plus*
 - The system crashes

Advantages of COMMIT and ROLLBACK Statements

With COMMIT and ROLLBACK statements, you can:

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

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 *iSQL*Plus*, without explicitly issuing COMMIT or ROLLBACK statements
- An automatic rollback occurs under an abnormal termination of *iSQL*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 within 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 inserted.
```

- 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.
```

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

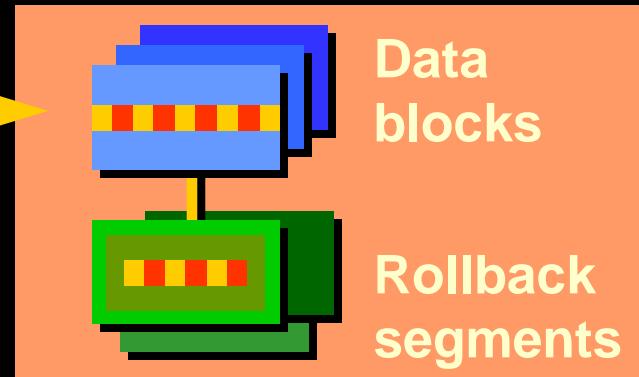
- **Read consistency guarantees a consistent view of the data at all times.**
- **Changes made by one user do not conflict with changes made by another user.**
- **Read consistency ensures that on the same data:**
 - Readers do not wait for writers
 - Writers do not wait for readers

Implementation of Read Consistency

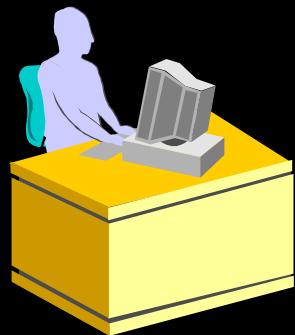
User A



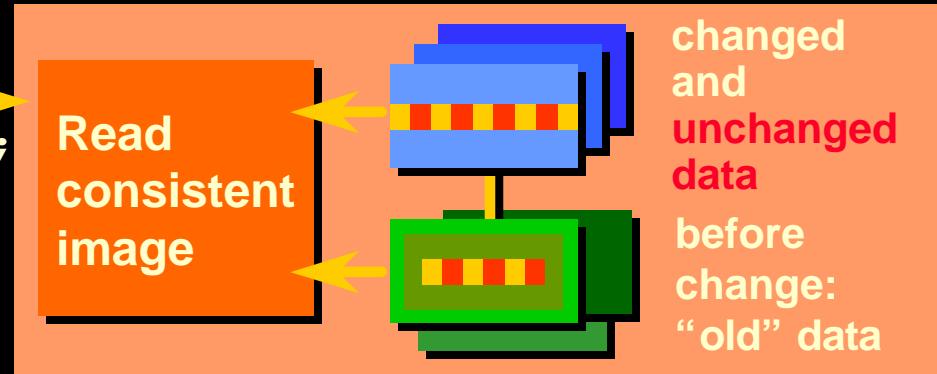
```
UPDATE employees  
SET salary = 7000  
WHERE last_name = 'Goyal';
```



User B



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



Locking

In an Oracle database, locks:

- Prevent destructive interaction between concurrent transactions
- Require no user action
- Use the lowest level of restrictiveness
- Are held for the duration of the transaction
- Are of two types: explicit locking and implicit locking

Implicit Locking

- **Two lock modes:**
 - **Exclusive:** Locks out other users
 - **Share:** Allows other users to access the server
- **High level of data concurrency:**
 - DML: Table share, row exclusive
 - Queries: No locks required
 - DDL: Protects object definitions
- **Locks held until commit or rollback**

Summary

In this lesson, you should have learned how to use DML statements and control transactions.

Statement	Description
INSERT	Adds a new row to the table
UPDATE	Modifies existing rows in the table
DELETE	Removes existing rows from the table
MERGE	Conditionally inserts or updates data in a table
COMMIT	Makes all pending changes permanent
SAVEPOINT	It is used to rollback 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**

9

Creating and Managing Tables

Objectives

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

- **Describe the main database objects**
- **Create tables**
- **Describe the data types that can be used when specifying column definition**
- **Alter table definitions**
- **Drop, rename, and truncate tables**

Database Objects

Object	Description
Table	Basic unit of storage; composed of rows and columns
View	Logically represents subsets of data from one or more tables
Sequence	Numeric value generator
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 to 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

The 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.**

The DEFAULT Option

- Specify a default value for a column during an `INSERT` operation.

```
... 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.

Creating Tables

- **Create the table.**

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

Table created.

- **Confirm creation of the table.**

```
DESCRIBE dept
```

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

Tables in the Oracle Database

- **User tables:**
 - Are a collection of tables created and maintained by the user
 - Contain user information
- **Data dictionary:**
 - Is a collection of tables created and maintained by the Oracle Server
 - Contain database information

Querying the Data Dictionary

- See the names of tables owned by the user.

```
SELECT    table_name  
FROM  user_tables;
```

- View distinct object types owned by the user.

```
SELECT DISTINCT object_type  
FROM user_objects;
```

- View tables, views, synonyms, and sequences owned by the user.

```
SELECT *  
FROM user_catalog;
```

Data Types

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

Datetime Data Types

Datetime enhancements with Oracle9i:

- New datetime data types have been introduced.
- New data type storage is available.
- Enhancements have been made to time zones and local time zone.

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 to 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.
- The **TIMESTAMP** data type is specified as follows:

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

TIMESTAMP WITH TIME ZONE Data Type

- **TIMESTAMP WITH TIME ZONE is a variant of TIMESTAMP that includes a time zone displacement in its value.**
- **The time zone displacement is the difference, in hours and minutes, between local time and UTC.**

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

TIMESTAMP WITH LOCAL TIME Data Type

- **TIMESTAMP WITH LOCAL TIME ZONE is another variant of TIMESTAMP that includes a time zone displacement in its value.**
- **Data stored in the database is normalized to the database time zone**
- **The time zone displacement is not stored as part of the column data; the server returns the data in the users' local session time zone.**
- **TIMESTAMP WITH LOCAL TIME ZONE data type is specified as follows:**

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

INTERVAL YEAR TO MONTH Data Type

- INTERVAL YEAR TO MONTH stores a period of time using the YEAR and MONTH datetime fields.

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

- Example:

```
INTERVAL '312-2' YEAR(3) TO MONTH
```

Indicates an interval of 312 years and 2 months

```
INTERVAL '312' YEAR(3)
```

Indicates 312 years and 0 months

```
INTERVAL '300' MONTH(3)
```

Indicates an interval of 300 months

Creating a Table by Using a Subquery Syntax

- 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

The 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**

The ALTER TABLE Statement

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

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

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

```
ALTER TABLE table
DROP         (column);
```

Adding a Column

DEPT80

EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE
149	Zlotkey	126000	29-JAN-00
174	Abel	132000	11-MAY-96
176	Taylor	103200	24-MAR-98

New column

JOB_ID

Add a new column to the DEPT80 table.

DEPT80

EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE	JOB_ID
149	Zlotkey	126000	29-JAN-00	
174	Abel	132000	11-MAY-96	
176	Taylor	103200	24-MAR-98	

Adding a Column

- Use the ADD clause to add columns.

```
ALTER TABLE dept80
ADD          (job_id VARCHAR2(9));
Table altered.
```

- The new column becomes the last column.

EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE	JOB_ID
149	Zlotkey	126000	29-JAN-00	
174	Abel	132000	11-MAY-96	
176	Taylor	103200	24-MAR-98	

Modifying a Column

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

```
ALTER TABLE dept80  
MODIFY      (last_name VARCHAR2(30));  
Table altered.
```

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

Dropping a Column

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

```
ALTER TABLE dept80
DROP COLUMN job_id;
Table altered.
```

The SET UNUSED Option

- You use the SET UNUSED option to mark one or more columns as unused.
- You use the DROP UNUSED COLUMNS option to remove the columns that are marked as unused.

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

OR

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

```
ALTER TABLE table
DROP UNUSED COLUMNS;
```

Dropping a Table

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

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

Changing the Name of an Object

- To change the name of a table, view, sequence, or synonym, execute the RENAME statement.

```
RENAME dept TO detail_dept;  
Table renamed.
```

- You must be the owner of the object.

Truncating a Table

- **The TRUNCATE TABLE statement:**
 - Removes all rows from a table
 - Releases the storage space used by that table

```
TRUNCATE TABLE detail_dept;
```

Table truncated.

- You cannot roll back row removal when using TRUNCATE.
- Alternatively, you can remove rows by using the DELETE statement.

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 use DDL statements to create, alter, drop, and rename tables.

Statement	Description
<code>CREATE TABLE</code>	Creates a table
<code>ALTER TABLE</code>	Modifies table structures
<code>DROP TABLE</code>	Removes the rows and table structure
<code>RENAME</code>	Changes the name of a table, view, sequence, or synonym
<code>TRUNCATE</code>	Removes all rows from a table and releases the storage space
<code>COMMENT</code>	Adds comments to a table or view

Practice 9 Overview

This practice covers the following topics:

- **Creating new tables**
- **Creating a new table by using the CREATE TABLE AS syntax**
- **Modifying column definitions**
- **Verifying that the tables exist**
- **Adding comments to tables**
- **Dropping tables**
- **Altering tables**

10

Including Constraints

Objectives

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

- **Describe constraints**
- **Create and maintain constraints**

What Are 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

- **Name a constraint or the Oracle server generates a name by using the `SYN_Cn` format.**
- **Create a constraint either:**
 - At the same time as the table is created, or
 - After the table has been created.
- **Define a constraint at the column or table level.**
- **View a constraint in the data dictionary.**

Defining Constraints

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

Defining Constraints

- **Column constraint level:**

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

- **Table constraint level:**

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

The NOT NULL Constraint

Ensures that null values are not permitted for the column

100	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000			
101	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000			100
102	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000			100
103	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000			102
104	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000			103

205	Higgins	SHIGGINS	515.123.8080	01-JUN-94	AC_MGR	5000			101
206	Gietz	WGIETZ	515.123.8181	07-JUN-94	AC_ACCOUNT	8300			205

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
null for this column.)**

The NOT NULL Constraint

Is defined at the column level

```
CREATE TABLE employees(
    employee_id      NUMBER(6),
    last_name        VARCHAR2(25) NOT NULL,           ← System named
    salary            NUMBER(8,2),
    commission_pct   NUMBER(2,2),
    hire_date         DATE
    CONSTRAINT emp_hire_date_nn
    NOT NULL,
    ...
)
```

System
named

User
named

The UNIQUE Constraint

EMPLOYEES

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

UNIQUE constraint

INSERT INTO

208	Smith	JSMITH
209	Smith	JSMITH

Allowed

Not allowed:
already exists

The UNIQUE Constraint

Is 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));
```

The 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

Not allowed
(null value)



INSERT INTO

	Public Accounting		1400
50	Finance	124	1500

Not allowed
(50 already exists)



The PRIMARY KEY Constraint

Is defined at either the table level or the column level

```
CREATE TABLE departments(
    department_id          NUMBER(4),
    department_name        VARCHAR2(30)
        CONSTRAINT dept_name_nn NOT NULL,
    manager_id              NUMBER(6),
    location_id              NUMBER(4),
        CONSTRAINT dept_id_pk PRIMARY KEY(department_id));
```

The FOREIGN KEY Constraint

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

PRIMARY
KEY

EMPLOYEES

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

FOREIGN
KEY

INSERT INTO

210	Ford	9
211	Ford	60

Not allowed
(9 does not
exist)

Allowed

The FOREIGN KEY Constraint

Is defined at either the table level or the column level

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

FOREIGN KEY Constraint Keywords

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

The 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),...
```

Adding a Constraint Syntax

Use the ALTER TABLE statement to:

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

```
ALTER TABLE table
ADD [CONSTRAINT constraint] type (column);
```

Adding a Constraint

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

```
ALTER TABLE      employees
ADD CONSTRAINT  emp_manager_fk
    FOREIGN KEY(manager_id)
    REFERENCES employees(employee_id);
Table altered.
```

Dropping a Constraint

- Remove the manager constraint from the EMPLOYEES table.

```
ALTER TABLE      employees  
DROP CONSTRAINT emp_manager_fk;
```

Table altered.

- Remove the PRIMARY KEY constraint on the DEPARTMENTS table and drop the associated FOREIGN KEY constraint on the EMPLOYEES.DEPARTMENT_ID column.

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

Table altered.

Disabling Constraints

- Execute the **DISABLE** clause of the **ALTER TABLE** statement to deactivate an integrity constraint.
- Apply the **CASCADE** option to disable dependent integrity constraints.

```
ALTER TABLE employees  
DISABLE CONSTRAINT emp_emp_id_pk CASCADE;  
Table altered.
```

Enabling Constraints

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

```
ALTER TABLE      employees  
ENABLE CONSTRAINT emp_emp_id_pk;  
Table altered.
```

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

Cascading Constraints

- The **CASCADE CONSTRAINTS clause** is used along with the **DROP COLUMN clause**.
- The **CASCADE CONSTRAINTS clause** drops all referential integrity constraints that refer to the primary and unique keys defined on the dropped columns.
- The **CASCADE CONSTRAINTS clause** also drops all multicolumn constraints defined on the dropped columns.

Cascading Constraints

Example

```
ALTER TABLE test1
DROP (pk) CASCADE CONSTRAINTS;
Table altered.
```

```
ALTER TABLE test1
DROP (pk, fk, col1) CASCADE CONSTRAINTS;
Table altered.
```

Viewing Constraints

Query the `USER_CONSTRAINTS` table to view all constraint definitions and names.

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

CONSTRAINT_NAME	C	SEARCH_CONDITION
EMP_LAST_NAME_NN	C	"LAST_NAME" IS NOT NULL
EMP_EMAIL_NN	C	"EMAIL" IS NOT NULL
EMP_HIRE_DATE_NN	C	"HIRE_DATE" IS NOT NULL
EMP_JOB_NN	C	"JOB_ID" IS NOT NULL
EMP_SALARY_MIN	C	salary > 0
EMP_EMAIL_UK	U	
EMP_EMP_ID_PK	P	
EMP_DEPT_FK	R	

Viewing the Columns Associated with Constraints

View the columns associated with the constraint names in the USER_CONS_COLUMNS view.

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

CONSTRAINT_NAME	COLUMN_NAME
EMP_DEPT_FK	DEPARTMENT_ID
EMP_EMAIL_NN	EMAIL
EMP_EMAIL_UK	EMAIL
EMP_EMP_ID_PK	EMPLOYEE_ID
EMP_HIRE_DATE_NN	HIRE_DATE
EMP_JOB_FK	JOB_ID
EMP_JOB_NN	JOB_ID
EMP_LAST_NAME_NN	LAST_NAME
EMP_MANAGER_FK	MANAGER_ID
EMP_SALARY_MIN	SALARY

Summary

In this lesson, you should have learned how to create constraints.

- **There are the following types of constraints:**
 - NOT NULL
 - UNIQUE
 - PRIMARY KEY
 - FOREIGN KEY
 - CHECK
- **You can query the USER_CONSTRAINTS table to view all constraint definitions and names.**

Practice 10 Overview

This practice covers the following topics:

- **Adding constraints to existing tables**
- **Adding more columns to a table**
- **Displaying information in data dictionary views**

11

Creating Views

Objectives

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

- **Describe a view**
- **Create, alter the definition of, and drop a view**
- **Retrieve data through a view**
- **Insert, update, and delete data through a view**
- **Create and use an inline view**
- **Perform top-*n* analysis**

Database Objects

Object	Description
Table	Basic unit of storage; composed of rows and columns
View	Logically represents subsets of data from one or more tables
Sequence	Generates primary key values
Index	Improves the performance of some queries
Synonym	Alternative name for an object

What Is a View?

EMPLOYEES Table

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000
							6000
							4200
							5800
							3500
							3100
							2800
							2600
							2500
							10500
							11000
							8600
							13000
							6000
							12000
							8300

EMPLOYEE_ID

LAST_NAME

SALARY

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

20 rows selected.

Why Use 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 within 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 a view, EMPVU80, that 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.
```

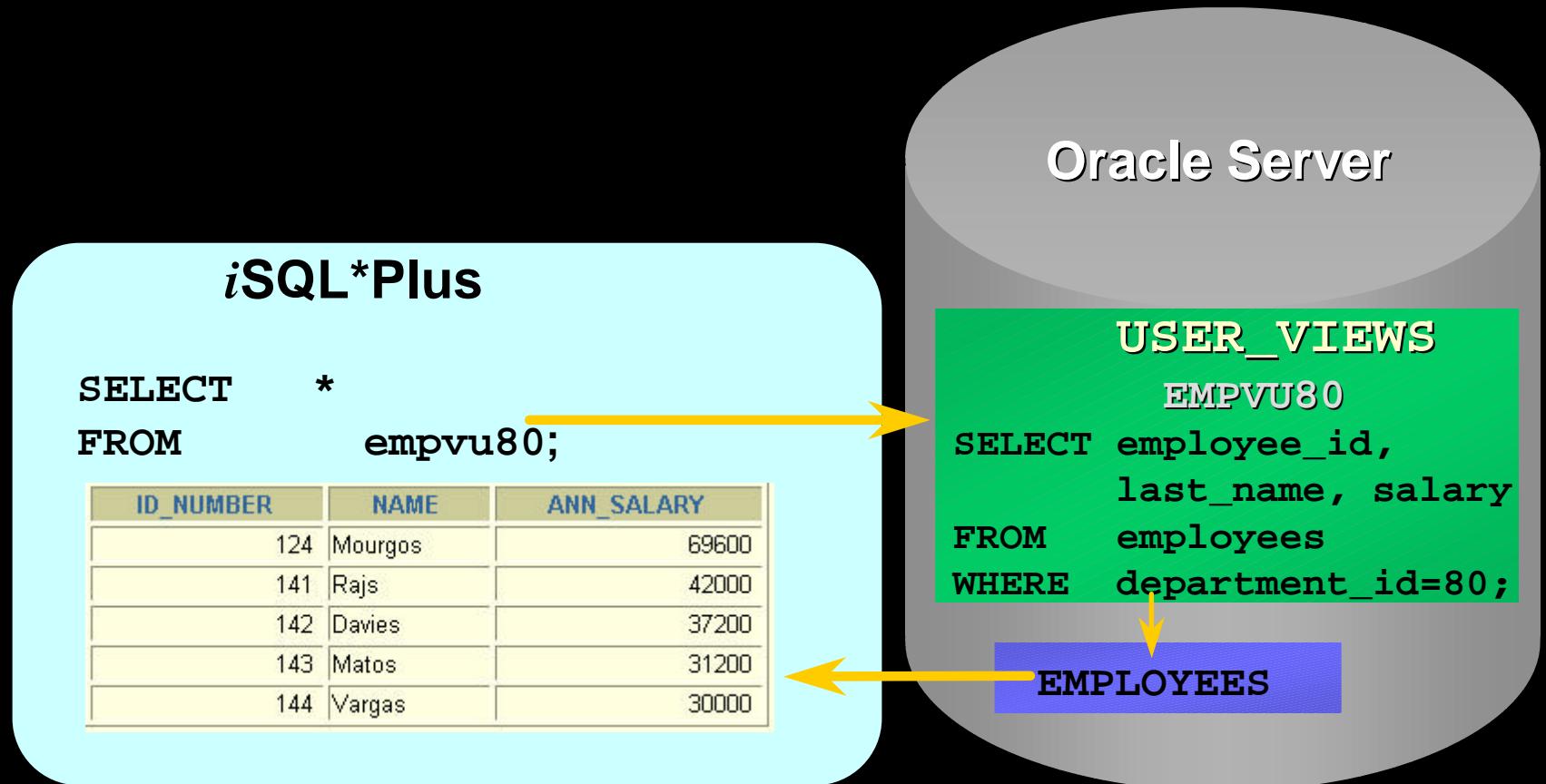
- Select the columns from this view by the given alias names.

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

Querying a View



Modifying a View

- **Modify the EMPVU80 view by using 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 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 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, departments d
  WHERE        e.department_id = d.department_id
  GROUP BY    d.department_name;
```

View created.



Rules for Performing DML Operations on a View

- You can 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 within 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 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.

Inline Views

- An **inline view** is a **subquery with an alias (or correlation name)** that you can use within a **SQL statement**.
- A **named subquery** in the **FROM clause of the main query** is an **example of an inline view**.
- An **inline view** is **not a schema object**.

Top-*n* Analysis

- **Top-*n* queries ask for the *n* largest or smallest values of a column. For example:**
 - What are the ten best selling products?
 - What are the ten worst selling products ?
- **Both largest values and smallest values sets are considered top-*n* queries.**

Performing Top-*n* Analysis

The high-level structure of a top-*n* analysis query is:

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

Example of Top-*n* Analysis

To display the top three earner names and salaries from the EMPLOYEES table.

```
SELECT ROWNUM as RANK, last_name, salary  
FROM  (SELECT last_name,salary FROM employees  
       ORDER BY salary DESC)  
WHERE ROWNUM <= 3;
```

RANK	LAST_NAME	SALARY
1	King	24000
2	Kochhar	17000
3	De Haan	17000

Summary

In this lesson you should have learned that a view is derived from data in other tables or other views and provides the following advantages:

- **Restricts database access**
- **Simplifies queries**
- **Provides data independence**
- **Provides multiple views of the same data**
- **Can be dropped without removing the underlying data**

Practice 11 Overview

This practice covers the following topics:

- **Creating a simple view**
- **Creating a complex view**
- **Creating a view with a check constraint**
- **Attempting to modify data in the view**
- **Displaying view definitions**
- **Removing views**

12

Other Database Objects

Objectives

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

- **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 and columns
View	Logically represents subsets of data from one or more tables
Sequence	Generates primary key values
Index	Improves the performance of some queries
Synonym	Alternative name for an object

What Is a Sequence?

A sequence:

- **Automatically generates unique numbers**
- **Is a sharable object**
- **Is typically used to create a primary key value**
- **Replaces application code**
- **Speeds up the efficiency of accessing sequence values when cached in memory**

The 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.

Confirming Sequences

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

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

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

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;
```

Using a Sequence

- **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
- **If the sequence was created with NOCACHE, view the next available value, by querying the USER_SEQUENCES 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.

Removing a Sequence

- Remove a sequence from the data dictionary by using the `DROP SEQUENCE` statement.
- Once removed, the sequence can no longer be referenced.

```
DROP SEQUENCE dept_deptid_seq;
```

Sequence dropped.

What Is an Index?

An index:

- Is a schema object
- Is 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 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.
```

When to Create an Index

You should create an index if:

- 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

When Not to Create an Index

It is usually not worth creating an index if:

- **The table is small**
- **The columns are not often used as a condition in the query**
- **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**

Confirming Indexes

- The **USER_INDEXES** data dictionary view contains the name of the index and its uniqueness.
- The **USER_IND_COLUMNS** view contains the index name, the table name, and the column name.

```
SELECT      ic.index_name, ic.column_name,  
            ic.column_position col_pos, ix.uniqueness  
FROM        user_indexes ix, user_ind_columns ic  
WHERE       ic.index_name = ix.index_name  
AND         ic.table_name = 'EMPLOYEES' ;
```

Function-Based Indexes

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

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

Index created.

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

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 upper_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

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

- Ease referring to a table 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:

- **Generate sequence numbers automatically by using a sequence generator**
- **View sequence information in the `USER_SEQUENCES` data dictionary table**
- **Create indexes to improve query retrieval speed**
- **View index information in the `USER_INDEXES` dictionary table**
- **Use synonyms to provide alternative names for objects**

Practice 12 Overview

This practice covers the following topics:

- **Creating sequences**
- **Using sequences**
- **Creating nonunique indexes**
- **Displaying data dictionary information about sequences and indexes**
- **Dropping indexes**

13

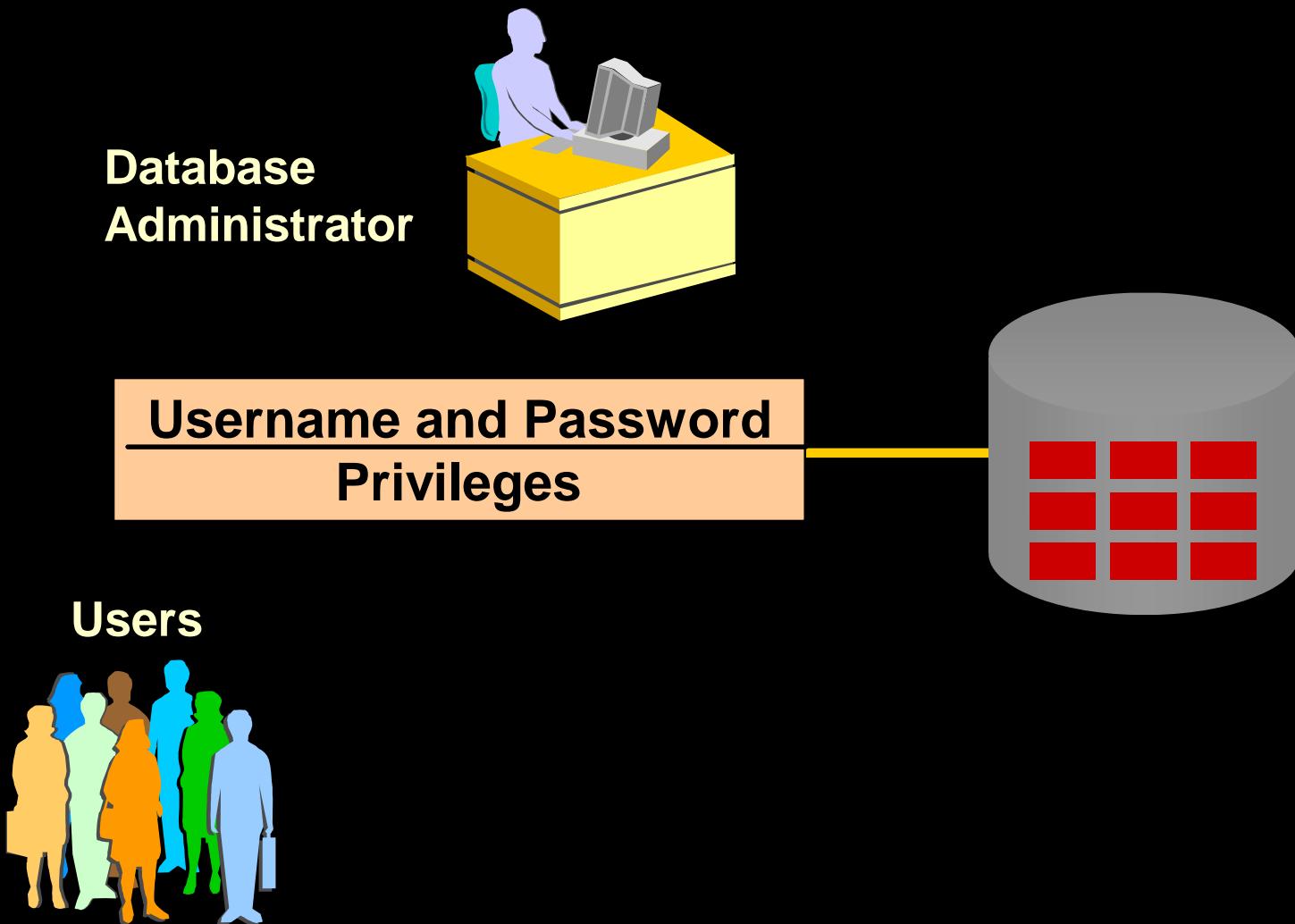
Controlling User Access

Objectives

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

- **Create users**
- **Create roles to ease setup and maintenance of the security model**
- **Use the GRANT and REVOKE statements to grant and revoke object privileges**
- **Create and access database links**

Controlling User Access



Privileges

- **Database security:**
 - System security
 - Data security
- **System privileges:** Gaining access to the database
- **Object privileges:** Manipulating the content of the database objects
- **Schemas:** Collections of objects, such as tables, views, and sequences

System Privileges

- More than 100 privileges are available.
- The database administrator has high-level system privileges for tasks such as:
 - Creating new users
 - Removing users
 - Removing tables
 - Backing up tables

Creating Users

The DBA creates users by using the CREATE USER statement.

```
CREATE USER user  
IDENTIFIED BY      password;
```

```
CREATE USER scott  
IDENTIFIED BY tiger;  
User created.
```

User System Privileges

- Once a user is created, the DBA can grant specific system privileges to a user.

```
GRANT privilege [, privilege...]
TO user [, user/ role, PUBLIC...];
```

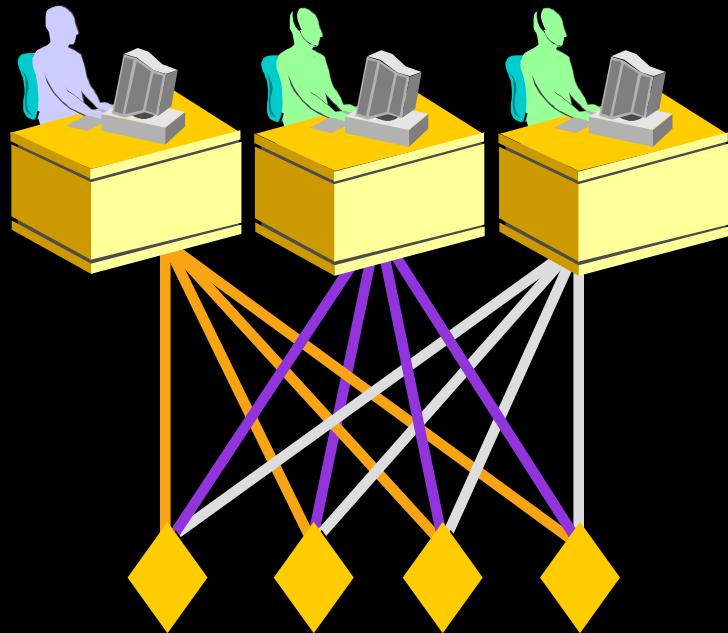
- An application developer, for example, may have the following system privileges:
 - CREATE SESSION
 - CREATE TABLE
 - CREATE SEQUENCE
 - CREATE VIEW
 - CREATE PROCEDURE

Granting System Privileges

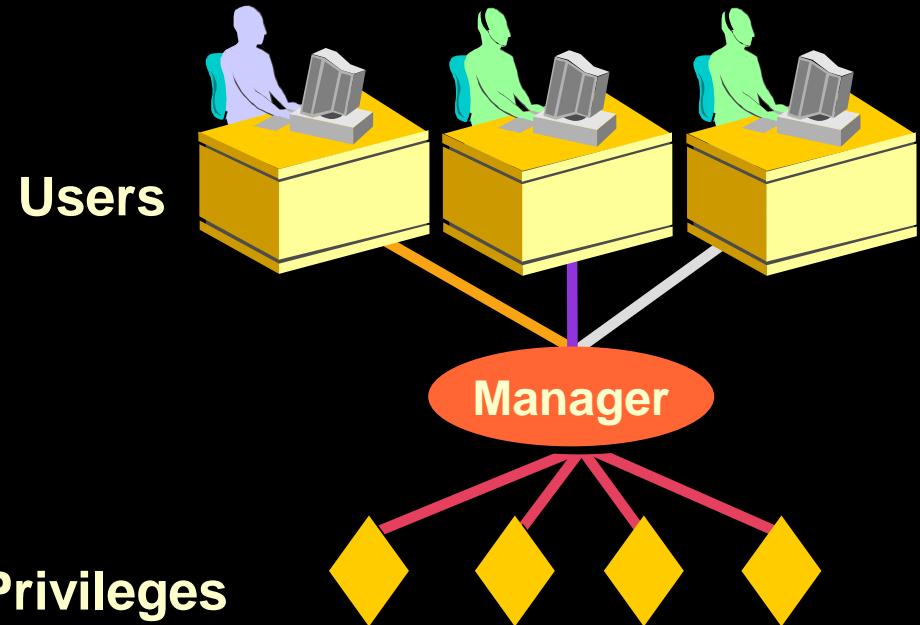
The DBA can grant a user specific system privileges.

```
GRANT  create session, create table,  
       create sequence, create view  
TO     scott;  
Grant succeeded.
```

What Is a Role?



Allocating privileges
without a role



Allocating privileges
with a role

Creating and Granting Privileges to a Role

- Create a role

```
CREATE ROLE manager;  
Role created.
```

- Grant privileges to a role

```
GRANT create table, create view  
TO manager;  
Grant succeeded.
```

- Grant a role to users

```
GRANT manager TO DEHAAN, KOCHHAR;  
Grant succeeded.
```

Changing Your Password

- The DBA creates your user account and initializes your password.
- You can change your password by using the ALTER USER statement.

```
ALTER USER scott  
IDENTIFIED BY lion;  
User altered.
```

Object Privileges

Object Privilege	Table	View	Sequence	Procedure
ALTER	ö		ö	
DELETE	ö	ö		
EXECUTE				ö
INDEX	ö			
INSERT	ö	ö		
REFERENCES	ö	ö		
SELECT	ö	ö	ö	
UPDATE	ö	ö		

Object Privileges

- Object privileges vary from object to object.
- An owner has all the privileges on the object.
- An owner can give specific privileges on that owner's object.

```
GRANT      object_priv [ (columns) ]  
ON         object  
TO         {user|role|PUBLIC}  
[WITH GRANT OPTION];
```

Granting Object Privileges

- Grant query privileges on the EMPLOYEES table.

```
GRANT select  
ON employees  
TO sue, rich;  
Grant succeeded.
```

- Grant privileges to update specific columns to users and roles.

```
GRANT update (department_name, location_id)  
ON departments  
TO scott, manager;  
Grant succeeded.
```

Using the WITH GRANT OPTION and PUBLIC Keywords

- Give a user authority to pass along privileges.

```
GRANT select, insert  
ON departments  
TO scott  
WITH GRANT OPTION;  
Grant succeeded.
```

- Allow all users on the system to query data from Alice's DEPARTMENTS table.

```
GRANT select  
ON alice.departments  
TO PUBLIC;  
Grant succeeded.
```

Confirming Privileges Granted

Data Dictionary View	Description
ROLE_SYS_PRIVS	System privileges granted to roles
ROLE_TAB_PRIVS	Table privileges granted to roles
USER_ROLE_PRIVS	Roles accessible by the user
USER_TAB_PRIVS_MADE	Object privileges granted on the user's objects
USER_TAB_PRIVS_REC'D	Object privileges granted to the user
USER_COL_PRIVS_MADE	Object privileges granted on the columns of the user's objects
USER_COL_PRIVS_REC'D	Object privileges granted to the user on specific columns
USER_SYS_PRIVS	Lists system privileges granted to the user

How to Revoke Object Privileges

- You use the REVOKE statement to revoke privileges granted to other users.
- Privileges granted to others through the WITH GRANT OPTION clause are also revoked.

```
REVOKE {privilege [, privilege...]|ALL}
ON      object
FROM    {user[, user...]|role|PUBLIC}
[CASCADE CONSTRAINTS];
```

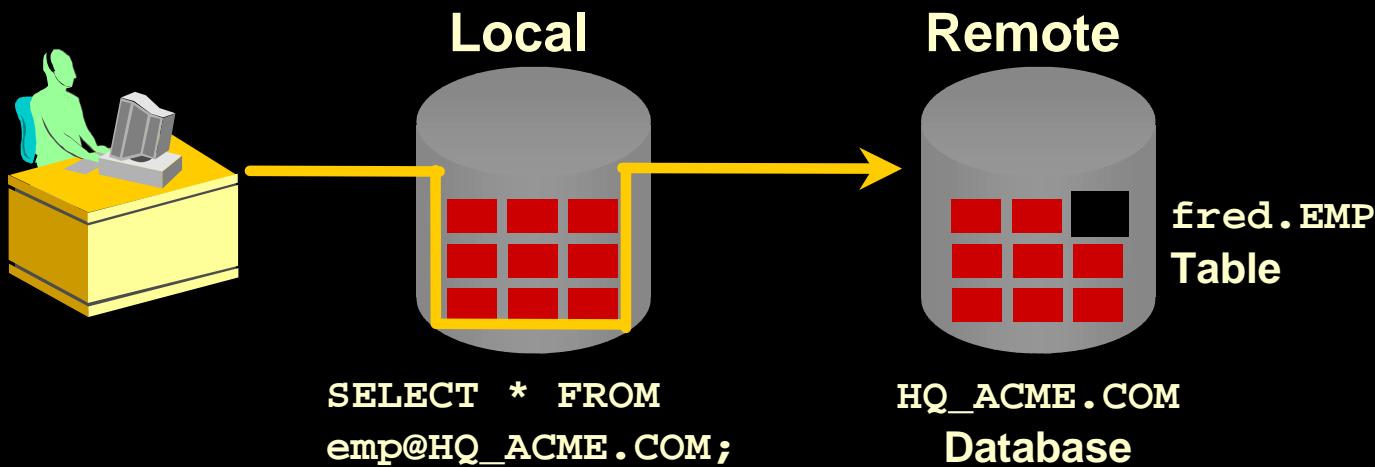
Revoking Object Privileges

As user Alice, revoke the SELECT and INSERT privileges given to user Scott on the DEPARTMENTS table.

```
REVOKE select, insert  
ON departments  
FROM scott;  
Revoke succeeded.
```

Database Links

A database link connection allows local users to access data on a remote database.



Database Links

- Create the database link.

```
CREATE PUBLIC DATABASE LINK hq.acme.com  
USING 'sales';  
Database link created.
```

- Write SQL statements that use the database link.

```
SELECT *  
FROM fred.emp@HQ.ACME.COM;
```

Summary

In this lesson you should have learned about DCL statements that control access to the database and database objects.

Statement	Action
CREATE USER	Creates a user (usually performed by a DBA)
GRANT	Gives other users privileges to access the your objects
CREATE ROLE	Creates a collection of privileges (usually performed by a DBA)
ALTER USER	Changes a user's password
REVOKE	Removes privileges on an object from users

Practice 13 Overview

This practice covers the following topics:

- **Granting other users privileges to your table**
- **Modifying another user's table through the privileges granted to you**
- **Creating a synonym**
- **Querying the data dictionary views related to privileges**

14

SQL Workshop

ORACLE®

Copyright © Oracle Corporation, 2001. All rights reserved.

Workshop Overview

This workshop covers:

- **Creating tables and sequences**
- **Modifying data in the tables**
- **Modifying table definitions**
- **Creating views**
- **Writing scripts containing SQL and iSQL*Plus commands**
- **Generating a simple report**

15

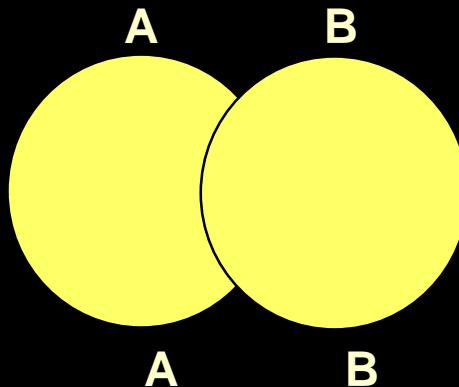
Using SET Operators

Objectives

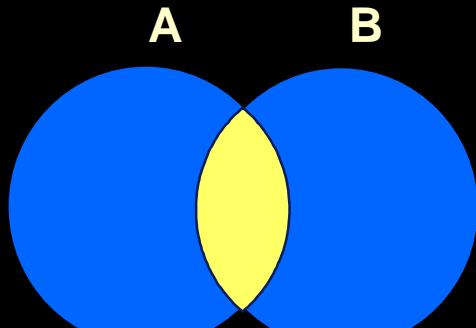
**After completing this lesson, you should be able
to do the following:**

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

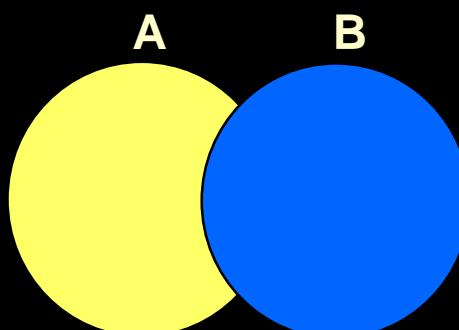
The SET Operators



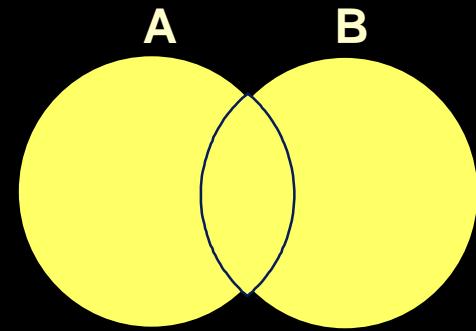
UNION/UNION ALL



INTERSECT



MINUS



Tables Used in This Lesson

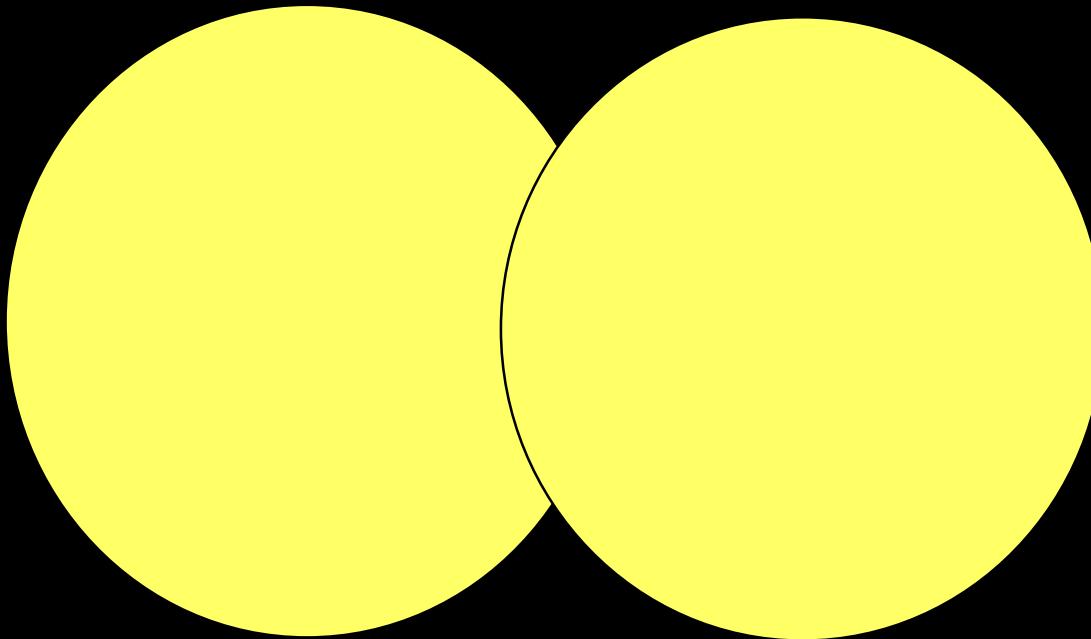
The tables used in this lesson are:

- **EMPLOYEES:** Provides details regarding all current employees
- **JOB_HISTORY:** When an employee switches jobs, the details of the start date and end date of the former job, the job identification number and department are recorded in this table

The UNION SET Operator

A

B



The UNION operator returns results from both queries after eliminating duplicates.

Using the UNION Operator

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

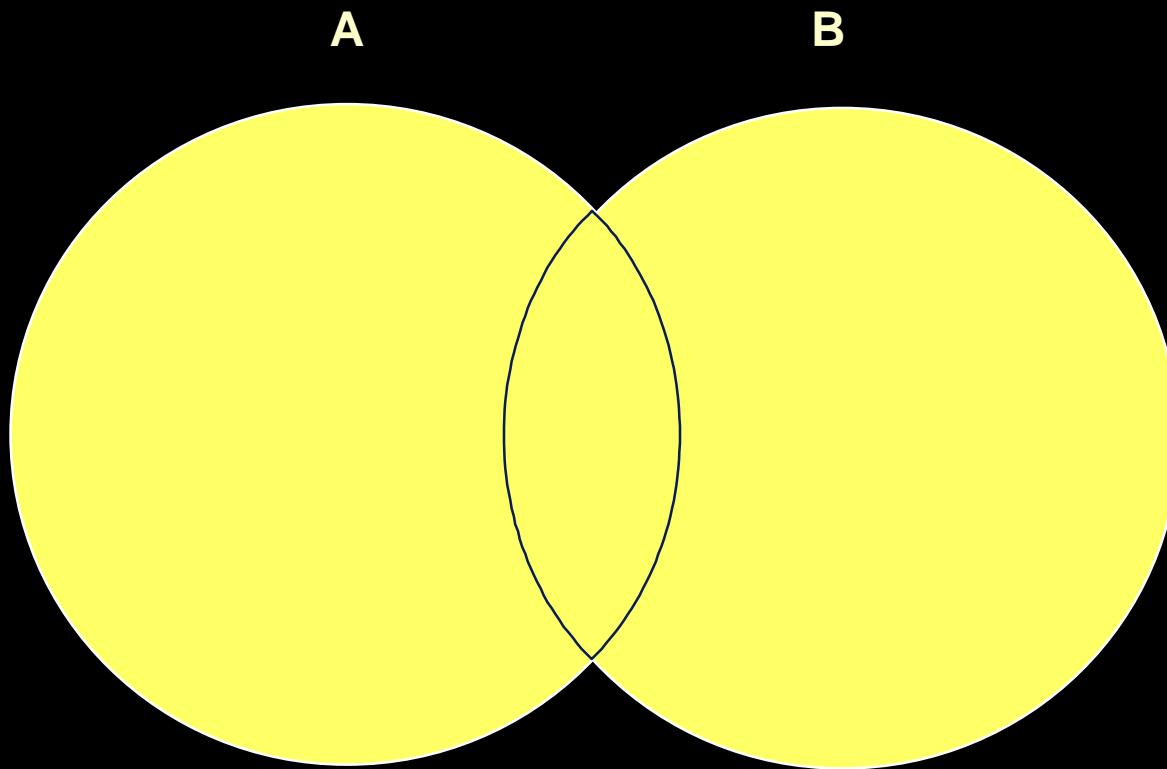
```
SELECT employee_id, job_id  
FROM   employees  
UNION  
SELECT employee_id, job_id  
FROM   job_history;
```

EMPLOYEE_ID	JOB_ID
100	AD_PRES
101	AC_ACCOUNT

101	AD_VP
178	SA_REP
200	AC_ACCOUNT
200	AD_ASST

28 rows selected.

The UNION ALL Operator



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

Using the UNION ALL Operator

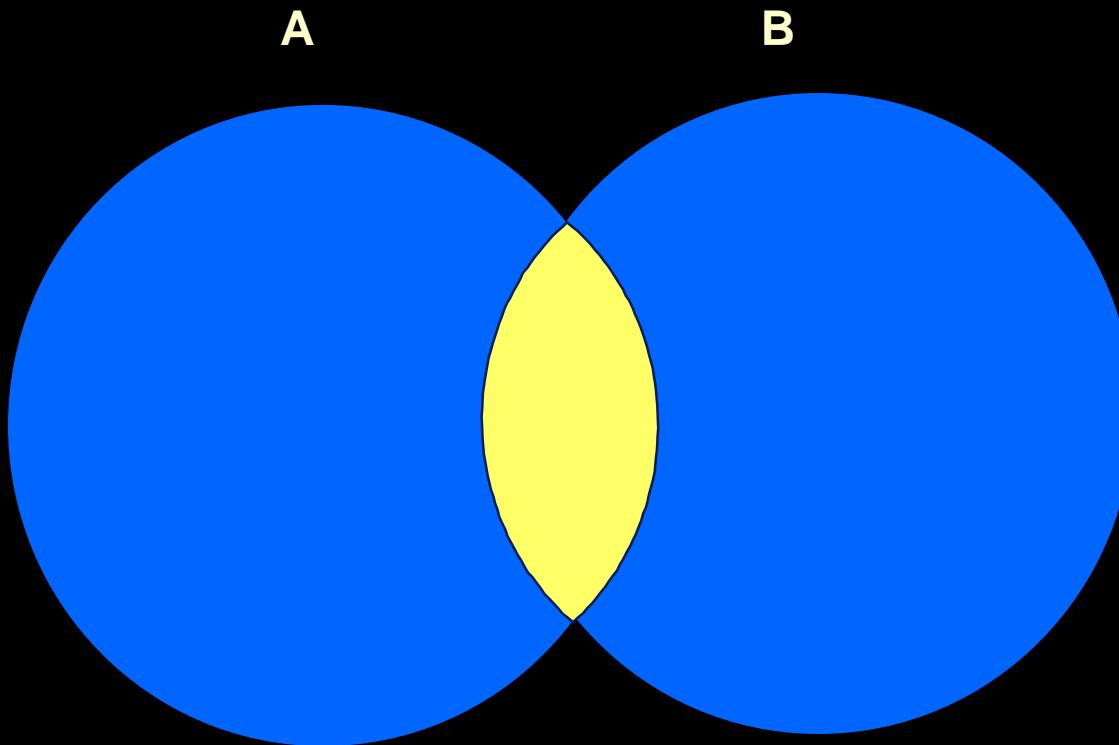
Display the current and previous departments of all employees.

```
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
174	SA_REP	80
176	SA_REP	80
176	SA_MAN	80
176	SA_REP	80
205	AC_MGR	110
206	AC_ACCOUNT	110

30 rows selected.

The INTERSECT Operator



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

Using the INTERSECT Operator

Display the employee IDs and job IDs of employees who are currently in a job title that they have held once before during their tenure with the company

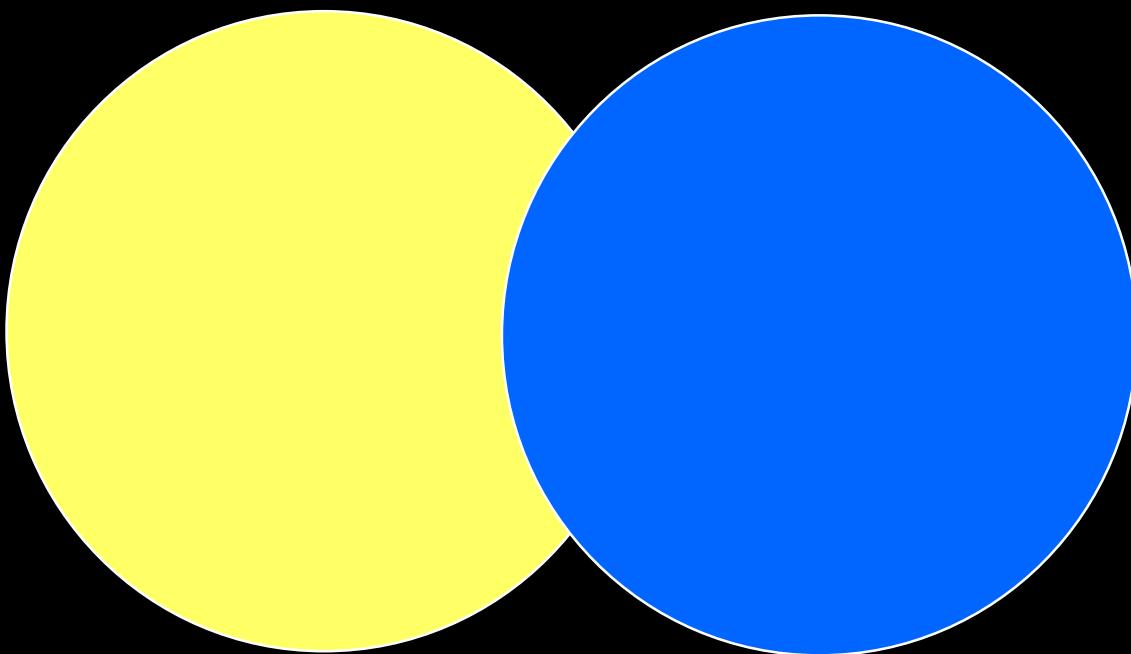
```
SELECT employee_id, job_id  
FROM   employees  
INTERSECT  
SELECT employee_id, job_id  
FROM   job_history;
```

EMPLOYEE_ID	JOB_ID
176	SA_REP
200	AD_ASST

The MINUS Operator

A

B



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

The MINUS Operator

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

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

EMPLOYEE_ID	JOB_ID
100	AD_PRES
101	AD_VP
102	AD_ASST

201	MK_MAN
202	MK_REP
205	AC_MGR
206	AC_ACCOUNT

18 rows selected.

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

190	1700	17-SEP-87
		17-FEB-96
		24-MAY-99

27 rows selected.



Matching the SELECT Statement

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
101	AC_HR	17000

205	AC_MGR	12000
206	AC_ACCOUNT	8300

30 rows selected.

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
FROM dual
UNION
SELECT 'the world to', 2
FROM dual
ORDER BY 2;
```

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

Summary

In this lesson, you should have learned the following:

- **UNION returns all distinct rows.**
- **UNION ALL returns all rows, including duplicates.**
- **INTERSECT returns all rows shared by both queries.**
- **MINUS returns all distinct rows selected by the first query but not by the second.**
- **ORDER BY can appear only at the very end of the statement.**

Practice 15 Overview

This practice covers the following topics:

- **Writing queries using the SET operators**
- **Discovering alternative join methods**

16

Oracle 9*i* Datetime Functions

ORACLE®

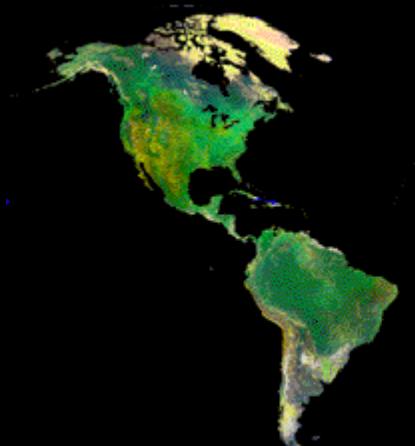
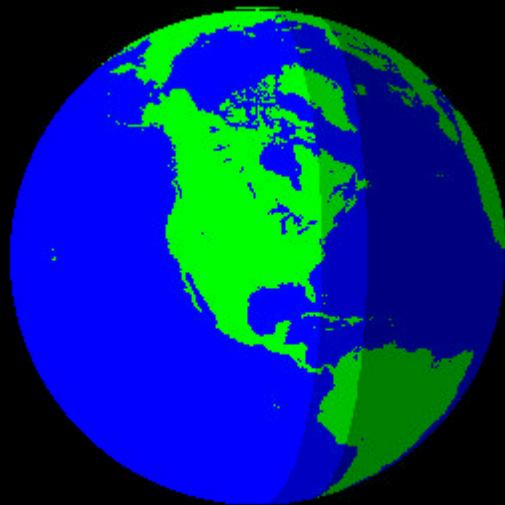
Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

**After completing this lesson, you should be able
use the following datetime functions:**

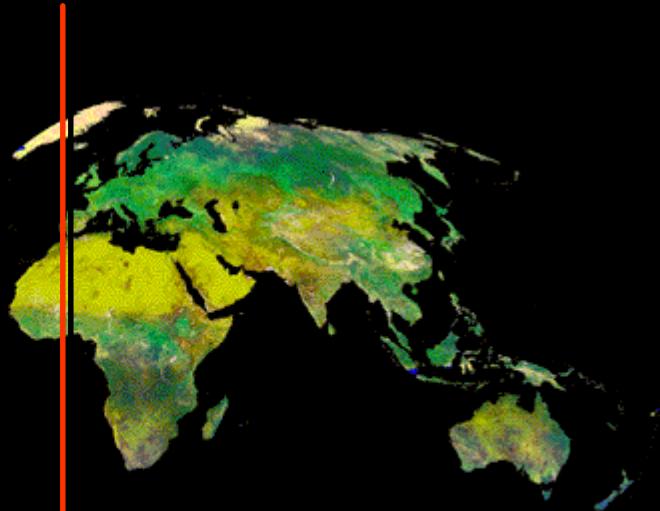
- CURRENT_DATE
- CURRENT_TIMESTAMP
- LOCALTIMESTAMP
- DBTIMEZONE
- SESSIONTIMEZONE
- EXTRACT
- FROM_TZ
- TO_TIMESTAMP
- TO_TIMESTAMP_TZ
- TO_YMINTERVAL
- TZ_OFFSET

TIME ZONES



GMT

0 °



0 °

Oracle 9*i* Datetime Support

- In Oracle9*i*, you can include the time zone in your date and time data, and provide support for fractional seconds.
- Three new data types are added to DATE:
 - TIMESTAMP
 - TIMESTAMP WITH TIME ZONE (TSTZ)
 - TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ)
- Oracle9*i* provides daylight savings support for datetime data types in the server.

CURRENT_DATE

```
ALTER SESSION SET NLS_DATE_FORMAT =
                      'DD-MON-YYYY HH24:MI:SS';
ALTER SESSION SET TIME_ZONE = '-5:0';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;
```

SESSIONTIMEZONE	CURRENT_DATE
-05:00	07-MAR-2001 03:31:50

```
ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;
```

SESSIONTIMEZONE	CURRENT_DATE
-08:00	07-MAR-2001 00:37:32

CURRENT_DATE is sensitive to the session time zone

CURRENT_TIMESTAMP

```
ALTER SESSION SET TIME_ZONE = '-5:0';
SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP FROM DUAL;
```

SESSIONTIMEZONE	CURRENT_TIMESTAMP
-05:00	07-MAR-01 03.42.04.799042 AM -05:00

```
ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP FROM DUAL;
```

SESSIONTIMEZONE	CURRENT_TIMESTAMP
-08:00	07-MAR-01 12.44.08.917054 AM -08:00

LOCALTIMESTAMP

```
ALTER SESSION SET TIME_ZONE = '-5:0';
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP FROM DUAL;
```

CURRENT_TIMESTAMP	LOCALTIMESTAMP
07-MAR-01 03.48.36.384601 AM -05:00	07-MAR-01 03.48.36.384601 AM

```
ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP FROM DUAL;
```

CURRENT_TIMESTAMP	LOCALTIMESTAMP
07-MAR-01 12.51.20.919127 AM -08:00	07-MAR-01 12.51.20.919127 AM

DBTIMEZONE and SESSIONTIMEZONE

```
SELECT DBTIMEZONE FROM DUAL;
```

DBTIME
+05:30

```
SELECT SESSIONTIMEZONE FROM DUAL;
```

SESSIONTIMEZONE
-08:00

EXTRACT

```
SELECT EXTRACT (YEAR FROM SYSDATE) FROM DUAL;
```

EXTRACT(YEARFROMSYSDATE)
2001

```
SELECT last_name, hire_date,  
       EXTRACT (MONTH FROM HIRE_DATE)  
FROM employees;  
WHERE manager id = 100;
```

LAST_NAME	HIRE_DATE	EXTRACT(MONTHFROMHIRE_DATE)
Kochhar	21-SEP-89	9
De Haan	13-JAN-93	1
Mourgos	16-NOV-99	11
Zlotkey	29-JAN-00	1
Hartstein	17-FEB-96	2

FROM_TZ

```
SELECT FROM_TZ(TIMESTAMP '2000-03-28 08:00:00','3:00')  
FROM DUAL;
```

FROM_TZ(TIMESTAMP'2000-03-2808:00:00','3:00')
28-MAR-00 08.00.00.000000000 AM +03:00

TO_TIMESTAMP and TO_TIMESTAMP_TZ

```
SELECT TO_TIMESTAMP ('2000-12-01 11:00:00',
                     'YYYY-MM-DD HH:MI:SS')
FROM DUAL;
```

TO_TIMESTAMP('2000-12-0111:00:00','YYYY-MM-DDHH:MI:SS')

01-DEC-00 11.00.00 AM

```
SELECT TO_TIMESTAMP_TZ('1999-12-01 11:00:00 -8:00',
                      'YYYY-MM-DD HH:MI:SS TZH:TZM')
FROM DUAL;
```

TO_TIMESTAMP_TZ('1999-12-0111:00:00-8:00','YYYY-MM-DDHH:MI:SSTZH:TZM')

01-DEC-99 11.00.00.000000000 AM -08:00

TO_YMINTERVAL

```
SELECT hire_date,  
       hire_date + TO_YMINTERVAL('01-02') AS  
          HIRE_DATE_YMININTERVAL  
FROM EMPLOYEES  
WHERE department_id = 20;
```

HIRE_DATE	HIRE_DATE_YMININTERV
17-FEB-1996 00:00:00	17-APR-1997 00:00:00
17-AUG-1997 00:00:00	17-OCT-1998 00:00:00

TZ_OFFSET

```
SELECT TZ_OFFSET('US/Eastern') FROM DUAL;
```

TZ_OFFSET
-05:00

```
SELECT TZ_OFFSET('Canada/Yukon') FROM DUAL;
```

TZ_OFFSET
-08:00

```
SELECT TZ_OFFSET('Europe/London') FROM DUAL;
```

TZ_OFFSET
+00:00

Summary

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

- **FROM_TZ**
- **TO_TIMESTAMP**
- **TO_TIMESTAMP_TZ**
- **TO_YMINTERVAL**
- **TZ_OFFSET**
- **CURRENT_DATE**
- **CURRENT_TIMESTAMP**
- **LOCALTIMESTAMP**
- **DBTIMEZONE**
- **SESSIONTIMEZONE**
- **EXTRACT**

Practice 16 Overview

This practice covers using the Oracle9*i* datetime functions.

17

Enhancements to the GROUP BY Clause

Objectives

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

- **Use the ROLLUP operation to produce subtotal values**
- **Use the CUBE operation to produce cross-tabulation values**
- **Use the GROUPING function to identify the row values created by ROLLUP or CUBE**
- **Use GROUPING SETS to produce a single result set**

Review of Group Functions

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

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

Example:

```
SELECT AVG(salary), STDDEV(salary),
       COUNT(commission_pct), MAX(hire_date)
FROM employees
WHERE job_id LIKE 'SA%';
```



Review of the GROUP BY Clause

Syntax:

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

Example:

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

Review of the HAVING Clause

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

- Use the HAVING clause to specify which groups are to be displayed.
- You further restrict the groups on the basis of a limiting condition.

GROUP BY with ROLLUP and CUBE Operators

- **Use ROLLUP or CUBE with GROUP BY to produce superaggregate rows by cross-referencing columns.**
- **ROLLUP grouping produces a results set containing the regular grouped rows and the subtotal values.**
- **CUBE grouping produces a results set containing the rows from ROLLUP and cross-tabulation rows.**

ROLLUP Operator

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

- ROLLUP is an extension to the GROUP BY clause.
- Use the ROLLUP operation to produce cumulative aggregates such as subtotals.

ROLLUP Operator Example

```
SELECT department_id, job_id, SUM(salary)  
FROM employees  
WHERE department_id < 60  
GROUP BY ROLLUP(department_id, job_id);
```

DEPARTMENT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
10		4400
20	MK_MAN	13000
20	MK_REP	6000
20		19000
50	ST_CLERK	11700
50	ST_MAN	5800
50		17500
		40900

9 rows selected.

1

2

3

CUBE Operator

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

- CUBE is an extension to the GROUP BY clause.
- You can use the CUBE operator to produce cross-tabulation values with a single SELECT statement.

CUBE Operator: Example

```
SELECT department_id, job_id, SUM(salary)  
FROM employees  
WHERE department_id < 60  
GROUP BY CUBE (department_id, job_id);
```

DEPARTMENT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
10		4400
20	MK_MAN	13000
20	MK_REP	6000
20		19000
50	ST_CLERK	11700
50	ST_MAN	5800
50		17500
	AD_ASST	4400
	MK_MAN	13000
	MK_REP	6000
	ST_CLERK	11700
	ST_MAN	5800
		40900

14 rows selected.

GROUPING Function

```
SELECT      [column,] group_function(column) . . . , GROUPING(expr)
FROM        table
[WHERE      condition]
[GROUP      BY      [ROLLUP][CUBE] group_by_expression]
[HAVING    having_expression];
[ORDER BY    column];
```

- The GROUPING function can be used with either the CUBE or ROLLUP operator.
- Using it, you can find the groups forming the subtotal in a row.
- Using it, you can differentiate stored NULL values from NULL values created by ROLLUP or CUBE.
- It returns 0 or 1.

GROUPING Function: Example

```
SELECT      department_id DEPTID, job_id JOB, SUM(salary),
GROUPING(department_id) GRP_DEPT, GROUPING(job_id) GRP_JOB
FROM        employees
WHERE       department_id < 50
GROUP BY   ROLLUP(department_id, job_id);
```

DEPTID	JOB	SUM(SALARY)	GRP_DEPT	GRP_JOB
10	AD_ASST	4400	0	0
10		4400	0	1
20	MK_MAN	13000	0	0
20	MK_REP	6000	0	0
20		19000	0	1
		23400	1	1

6 rows selected.

GROUPING SETS

- GROUPING SETS are a further extension of the GROUP BY clause.
- You can use GROUPING SETS to define multiple groupings in the same query.
- The Oracle Server computes all groupings specified in the GROUPING SETS clause and combines the results of individual groupings with a UNION ALL operation.
- Grouping set efficiency:
 - Only one pass over the base table is required.
 - There is no need to write complex UNION statements.
 - The more elements the GROUPING SETS have, the higher the performance benefit is.

GROUPING SETS: Example

```
SELECT department_id, job_id, manager_id, avg(salary)
FROM employees
GROUP BY GROUPING SETS
((department_id,job_id), (job_id,manager_id));
```

DEPARTMENT_ID	JOB_ID	MANAGER_ID	Avg(Salary)
10	AD_ASST		4400
20	MK_MAN		13000
20	MK_REP		6000
50	ST_CLERK		2925
50	ST_MAN		5800

1

DEPARTMENT_ID	JOB_ID	MANAGER_ID	Avg(Salary)
10	MK_MAN	100	13000
20	MK_REP	201	6000
20	SA_MAN	100	10500
20	SA_REP	149	8866.66667
50	ST_CLERK	124	2925
50	ST_MAN	100	5800

2

26 rows selected.

Composite Columns

- A composite column is a collection of columns that are treated as a unit.
`ROLLUP (a,(b,c) , d)`
- To specify composite columns, in the GROUP BY clause you group columns within parentheses so that the Oracle server treats them as a unit while computing ROLLUP or CUBE operations.
- When used with ROLLUP or CUBE, composite columns would mean skipping aggregation across certain levels.

Composite Columns: Example

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

DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
10	AD_ASST	101	4400
10			4400
20	MK_MAN	100	13000
20	MK_REP	201	6000
20			19000
50	ST_CLERK	124	11700
50	ST_MAN	100	5800
50			17500
10	AC_MGR	111	12000
110			20300
	SA_REP	149	7000
			7000
			175500

23 rows selected.

Concatenated Groupings

- Concatenated groupings offer a concise way to generate useful combinations of groupings.
- To specify concatenated grouping sets, you separate multiple grouping sets, ROLLUP, and CUBE operations with commas so that the Oracle Server combines them into a single GROUP BY clause.
- The result is a cross-product of groupings from each grouping set.

```
GROUP BY GROUPING SETS(a, b), GROUPING SETS(c, d)
```

Concatenated Groupings Example

```
SELECT department_id, job_id, manager_id, SUM(salary)  
FROM employees  
GROUP BY department_id, ROLLUP(job_id), CUBE(manager_id);
```

DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
10	AD_ASST	101	4400
20	MK_MAN	100	13000
20	MK_REP	201	6000
50	ST_MAN	100	5800
10		101	4400
		201	6000
20			6000
10	AD_ASST		4400
10			4400

1

2

3

4

49 rows selected.

Summary

In this lesson, you should have learned how to:

- **Use the ROLLUP operation to produce subtotal values**
- **Use the CUBE operation to produce cross-tabulation values**
- **Use the GROUPING function to identify the row values created by ROLLUP or CUBE**
- **Use the GROUPING SETS syntax to define multiple groupings in the same query.**
- **Use the GROUP BY clause, to combine expressions in various ways:**
 - **Composite columns**
 - **Concatenated grouping sets**

Practice 17 Overview

This practice covers the following topics:

- **Using the ROLLUP operator**
- **Using the CUBE operator**
- **Using the GROUPING function**
- **Using GROUPING SETS**

18

Advanced Subqueries

Objectives

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

- **Write a multiple-column subquery**
- **Describe and explain the behavior of subqueries when null values are retrieved**
- **Write a subquery in a FROM clause**
- **Use scalar subqueries in SQL**
- **Describe the types of problems that can be solved with correlated subqueries**
- **Write correlated subqueries**
- **Update and delete rows using correlated subqueries**
- **Use the EXISTS and NOT EXISTS operators**
- **Use the WITH clause**

What Is a Subquery?

A **subquery** is a **SELECT statement embedded in a clause of another SQL statement**.

Main query →

```
SELECT ...
FROM ...
WHERE ...
```

```
( SELECT ...
  FROM ...
 WHERE ... )
```

← Subquery

Subqueries

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

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

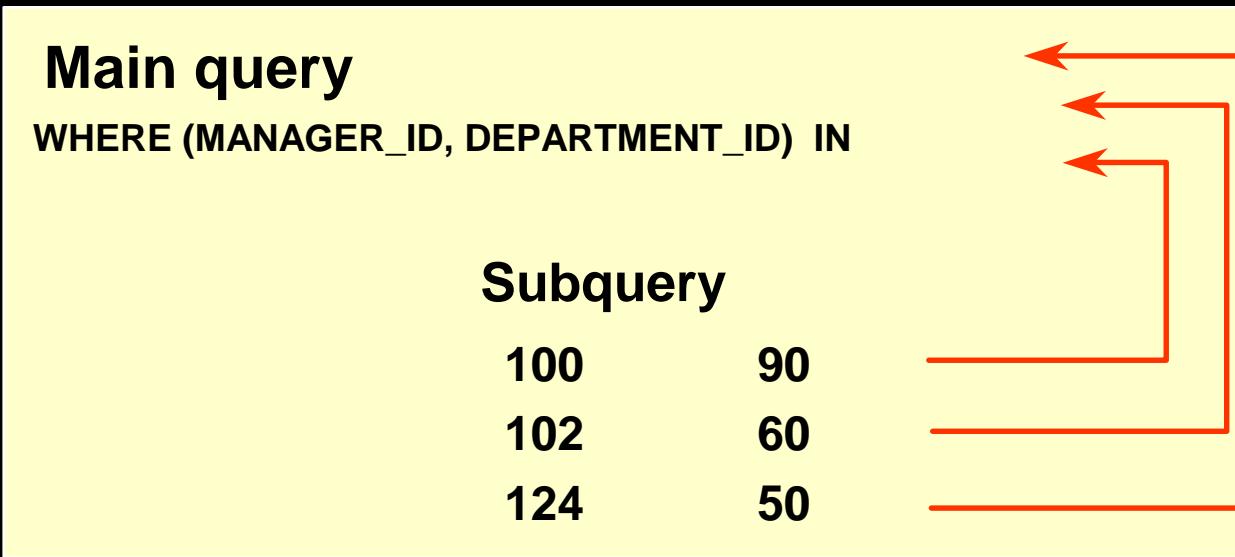
Using a Subquery

```
SELECT last_name
FROM   employees
WHERE  salary >          10500
       (SELECT salary
        FROM   employees
        WHERE  employee_id = 149);
```

LAST_NAME
King
Kochhar
De Haan
Abel
Hartstein
Higgins

6 rows selected.

Multiple-Column Subqueries



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

Column Comparisons

Column comparisons in a multiple-column subquery can be:

- **Pairwise comparisons**
- **Nonpairwise comparisons**

Pairwise Comparison Subquery

Display the details of the employees who are managed by the same manager *and* work in the same department as the employees with EMPLOYEE_ID 178 or 174.

```
SELECT employee_id, manager_id, department_id
FROM   employees
WHERE  (manager_id, department_id) IN
        (SELECT manager_id, department_id
         FROM   employees
         WHERE  employee_id IN (178,174))
AND    employee_id NOT IN (178,174);
```

Nonpairwise Comparison Subquery

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

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

Using a Subquery in the FROM Clause

```
SELECT a.last_name, a.salary, a.department_id, b.salavg
FROM   employees a, (SELECT department_id,
                           AVG(salary) salavg
                        FROM   employees
                        GROUP BY department_id) b
WHERE  a.department_id = b.department_id
AND    a.salary > b.salavg;
```

LAST_NAME	SALARY	DEPARTMENT_ID	SALAVG
Hartstein	13000	20	9500
Mourgos	5800	50	3500
Hunold	9000	60	6400
Zlotkey	10500	80	10033.3333
Abel	11000	80	10033.3333
King	24000	90	19333.3333
Higgins	12000	110	10150

7 rows selected.



Scalar Subquery Expressions

- A scalar subquery expression is a subquery that returns exactly one column value from one row.
- Scalar subqueries were supported in Oracle8i only in a limited set of cases, For example :
 - SELECT statement (FROM, WHERE clauses)
 - VALUES list of an INSERT statement
- In Oracle9i, scalar subqueries can be used in:
 - Condition and expression part of DECODE and CASE
 - All clauses of SELECT except GROUP BY

Scalar Subqueries: Examples

Scalar Subqueries in CASE Expressions

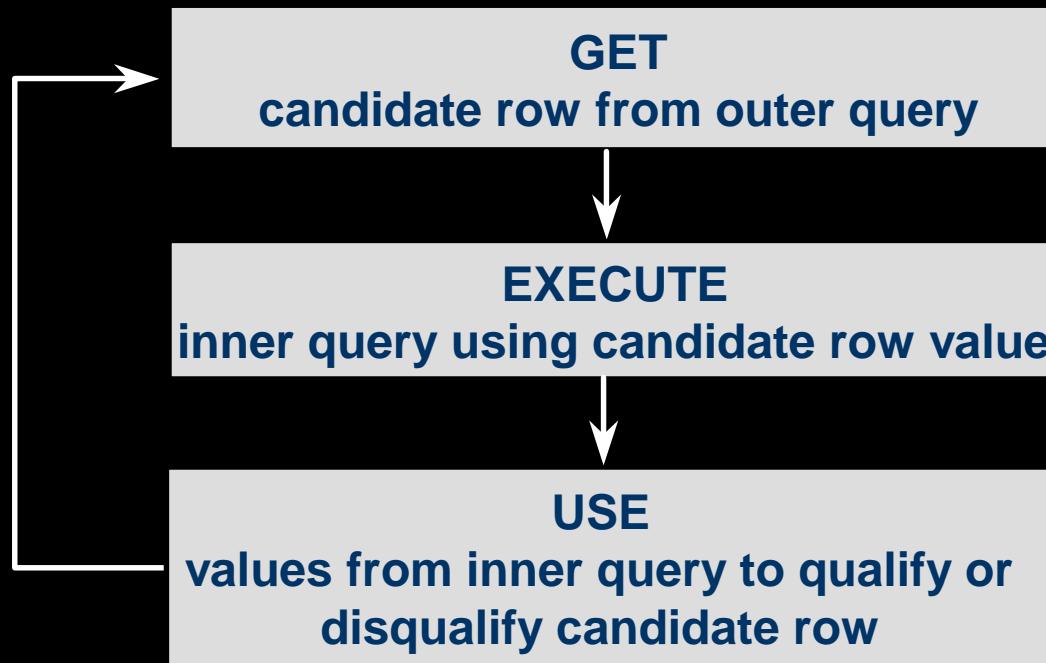
```
SELECT employee_id, last_name,  
       (CASE  
           WHEN department_id = 20 ←  
               (SELECT department_id FROM departments  
                WHERE location_id = 1800)  
           THEN 'Canada' ELSE 'USA' END) location  
  FROM employees;
```

Scalar Subqueries in ORDER BY Clause

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

Correlated Subqueries

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



Correlated Subqueries

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

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

Using Correlated Subqueries

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

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

LAST_NAME	SALARY	DEPARTMENT_ID
King	24000	90
Hunold	9000	60
Mourgos	5800	50
Zlotkey	10500	80
Abel	11000	80
Hartstein	13000	20
Higgins	12000	110

7 rows selected.

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

Using Correlated Subqueries

Display details of those employees who have switched jobs at least twice.

```
SELECT e.employee_id, last_name, e.job_id  
FROM   employees e  
WHERE  2 <= (SELECT COUNT(*)  
              FROM   job_history  
              WHERE  employee_id = e.employee_id);
```

EMPLOYEE_ID	LAST_NAME	JOB_ID
101	Kochhar	AD_VP
176	Taylor	SA_REP
200	Whalen	AD_ASST

Using the EXISTS Operator

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

Using the EXISTS Operator

Find employees who have at least one person reporting to them.

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

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
124	Mourgos	ST_MAN	50
149	Zlotkey	SA_MAN	80
201	Hartstein	MK_MAN	20
205	Higgins	AC_MGR	110

8 rows selected.



Using the NOT EXISTS Operator

Find all departments that do not have any employees.

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

DEPARTMENT_ID	DEPARTMENT_NAME
190	Contracting

Correlated UPDATE

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

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

Correlated UPDATE

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

```
ALTER TABLE employees  
ADD(department_name VARCHAR2(14));
```

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

Correlated DELETE

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

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

Correlated DELETE

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

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

The WITH Clause

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

WITH Clause: Example

Using the WITH clause, write a query to display the department name and total salaries for those departments whose total salary is greater than the average salary across departments.

WITH Clause: Example

```
WITH
dept_costs AS (
    SELECT department_name, SUM(salary) AS dept_total
    FROM employees, departments
    WHERE employees.department_id =
          departments.department_id
    GROUP BY department_name),
     avg_cost AS
    (SELECT SUM(dept_total)/COUNT(*) AS dept_avg
     FROM dept_costs)
SELECT * FROM dept_costs
WHERE dept_total >
(SELECT dept_avg FROM dept_avg)
ORDER BY department_name;
```



Summary

In this lesson, you should have learned the following:

- **A multiple-column subquery returns more than one column.**
- **Multiple-column comparisons can be pairwise or nonpairwise.**
- **A multiple-column subquery can also be used in the `FROM` clause of a `SELECT` statement.**
- **Scalar subqueries have been enhanced in Oracle 9*i*.**

Summary

- **Correlated subqueries are useful whenever a subquery must return a different result for each candidate row.**
- **The EXISTS operator is a Boolean operator that tests the presence of a value.**
- **Correlated subqueries can be used with SELECT, UPDATE, and DELETE statements.**
- **You can use the WITH clause to use the same query block in a SELECT statement when it occurs more than once**

Practice 18 Overview

This practice covers the following topics:

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

19

Hierarchical Retrieval

Objectives

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

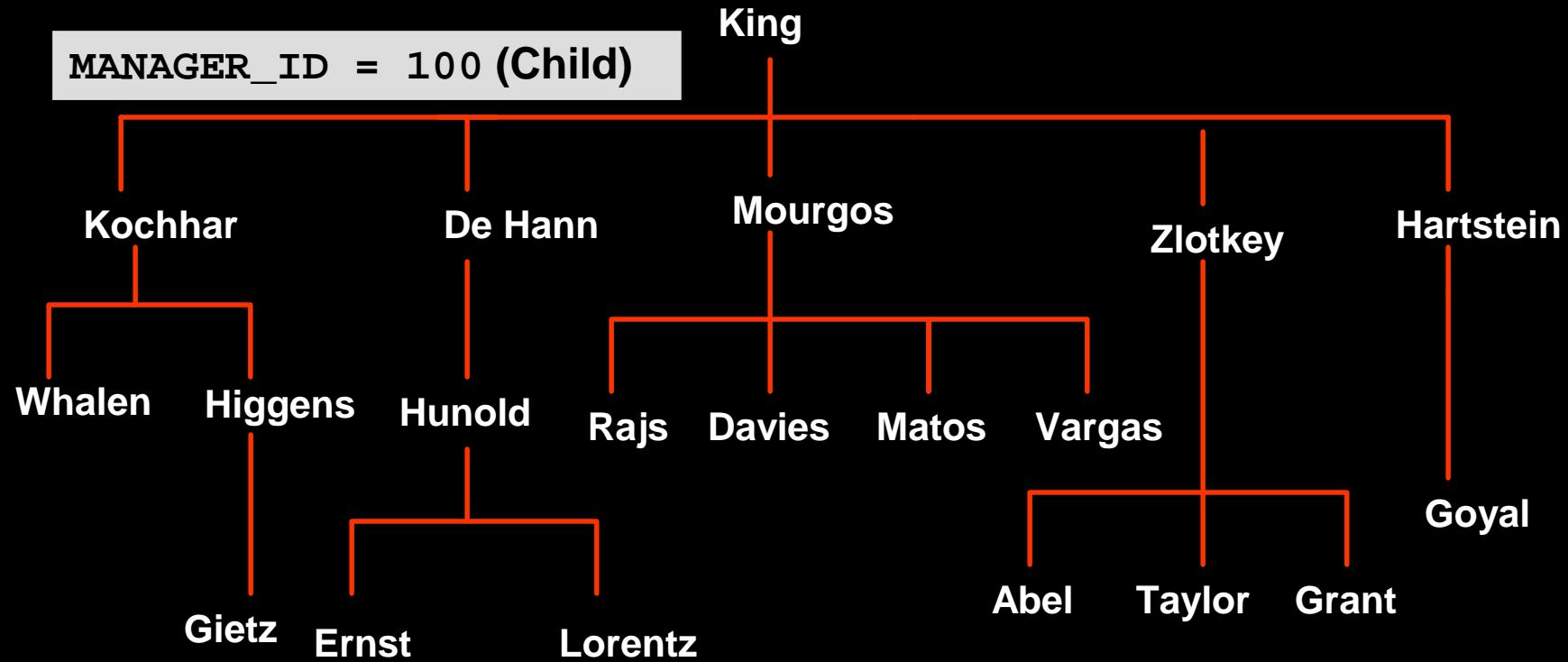
- Interpret the concept of a hierarchical query
- Create a tree-structured report
- Format hierarchical data
- Exclude branches from the tree structure

Sample Data from the EMPLOYEES Table

EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
100	King	AD_PRES	
101	Kochhar	AD_VP	100
102	De Haan	AD_VP	100
103	Hunold	IT_PROG	102
104	Ernst	IT_PROG	103
107	Lorentz	IT_PROG	103
124	Mourgos	ST_MAN	100
141	Rajs	ST_CLERK	124
142	Davies	ST_CLERK	124
143	Matos	ST_CLERK	124
144	Vargas	ST_CLERK	124
149	Zlotkey	SA_MAN	100
174	Abel	SA_REP	149
176	Taylor	SA_REP	149
178	Grant	SA_REP	149
200	Whalen	AD_ASST	101
201	Hartstein	MK_MAN	100
202	Fay	MK_REP	201
205	Higgins	AC_MGR	101
206	Gietz	AC_ACCOUNT	205

Natural Tree Structure

EMPLOYEE_ID = 100 (Parent)



Hierarchical Queries

```
SELECT [LEVEL], column, expr...
  FROM table
  [WHERE condition(s)]
  [START WITH condition(s)]
  [CONNECT BY PRIOR condition(s)];
```

WHERE *condition*:

```
expr comparison_operator expr
```

Walking the Tree

Starting Point

- Specifies the condition that must be met
- Accepts any valid condition

```
START WITH column1 = value
```

- Using the EMPLOYEES table, start with the employee whose last name is Kochhar.

```
...START WITH last_name = 'Kochhar'
```

Walking the Tree

```
CONNECT BY PRIOR column1 = column2
```

Walk from the top down using the EMPLOYEES table

```
... CONNECT BY PRIOR employee_id = manager_id
```

Direction

Top down → **Column1 = Parent Key**
Column2 = Child Key

Bottom up → **Column1 = Child Key**
Column2 = Parent Key

Walking the Tree: From the Bottom Up

```
SELECT employee_id, last_name, job_id, manager_id  
FROM   employees  
START WITH employee_id = 101  
CONNECT BY PRIOR manager_id = employee_id;
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
101	Kochhar	AD_VP	100
100	King	AD_PRES	

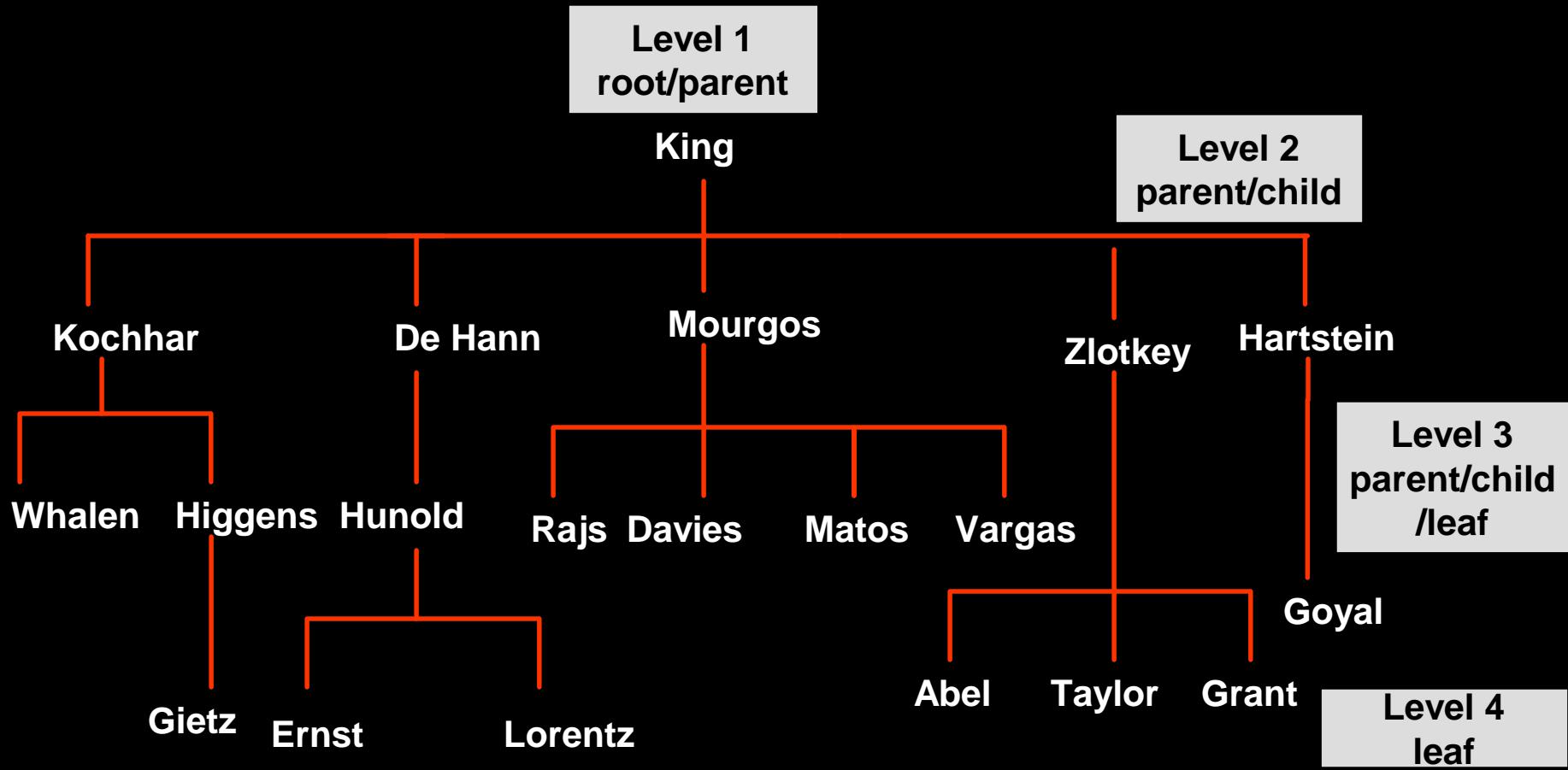
Walking the Tree: From the Top Down

```
SELECT  last_name || ' reports to ' ||
PRIOR   last_name "Walk Top Down"
FROM    employees
START   WITH last_name = 'King'
CONNECT BY PRIOR employee_id = manager_id;
```

Walk Top Down
King reports to
Hartstein reports to King
Fay reports to Hartstein
Kochhar reports to King
Whalen reports to Kochhar
Matos reports to Mourgos
Vargas reports to Mourgos
Zlotkey reports to King
Abel reports to Zlotkey
Taylor reports to Zlotkey
Grant reports to Zlotkey
20 rows selected.



Ranking Rows with the LEVEL Pseudocolumn



Formatting Hierarchical Reports Using LEVEL and LPAD

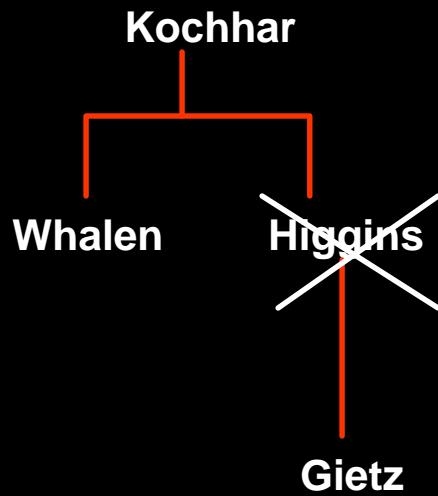
Create a report displaying company management levels, beginning with the highest level and indenting each of the following levels.

```
COLUMN org_chart FORMAT A12
SELECT LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-2, '_')
       AS org_chart
FROM   employees
START WITH last_name='King'
CONNECT BY PRIOR employee_id=manager_id
```

Pruning Branches

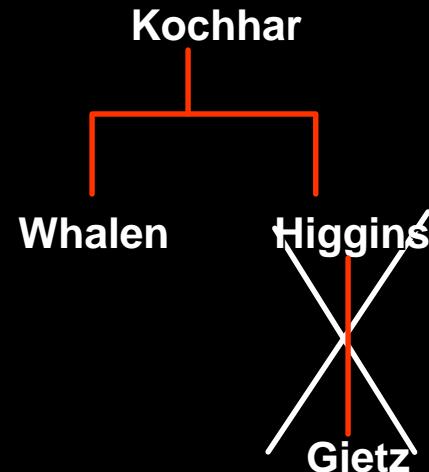
Use the WHERE clause
to eliminate a node.

```
WHERE last_name != 'Higgins'
```



Use the CONNECT BY clause
to eliminate a branch.

```
CONNECT BY PRIOR  
employee_id = manager_id  
AND last_name != 'Higgins'
```



Summary

In this lesson, you should have learned the following:

- You can use hierarchical queries to view a hierarchical relationship between rows in a table.
- You specify the direction and starting point of the query.
- You can eliminate nodes or branches by pruning.

Practice 19 Overview

This practice covers the following topics:

- **Distinguishing hierarchical queries from nonhierarchical queries**
- **Performing tree walks**
- **Producing an indented report by using the LEVEL pseudocolumn**
- **Pruning the tree structure**
- **Sorting the output**

20 Oracle 9*i* Extensions to DML and DDL Statements

ORACLE®

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Describe the features of multitable inserts**
- **Use the following types of multitable inserts**
 - **Unconditional INSERT**
 - **Pivoting INSERT**
 - **Conditional ALL INSERT**
 - **Conditional FIRST INSERT**
- **Create and use external tables**
- **Name the index at the time of creating a primary key constraint**

Review of the INSERT Statement

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

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

- Only one row is inserted at a time with this syntax.

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

Review of the UPDATE Statement

- **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.
- Specific row or rows are modified if you specify the WHERE clause.

```
UPDATE employees
SET     department_id = 70
WHERE   employee_id = 142;
1 row updated.
```

Overview of Multitable INSERT Statements

- The **INSERT...SELECT statement can be used to insert rows into multiple tables as part of a single DML statement.**
- **Multitable INSERT statements can be used in data warehousing systems to transfer data from one or more operational sources to a set of target tables.**
- **They provide significant performance improvement over:**
 - Single DML versus multiple `INSERT.. SELECT` statements
 - Single DML versus a procedure to do multiple inserts using `IF... THEN` syntax



Types of Multitable INSERT Statements

Oracle9i introduces the following types of multitable insert statements:

- **Unconditional INSERT**
- **Conditional ALL INSERT**
- **Conditional FIRST INSERT**
- **Pivoting INSERT**

Multitable INSERT Statements

Syntax

```
INSERT [ALL] [conditional_insert_clause]
[insert_into_clause values_clause] (subquery)
```

conditional_insert_clause

```
[ALL] [FIRST]
[WHEN condition THEN] [insert_into_clause values_clause]
[ELSE] [insert_into_clause values_clause]
```

Unconditional INSERT ALL

- Select the **EMPLOYEE_ID**, **HIRE_DATE**, **SALARY**, and **MANAGER_ID** values from the **EMPLOYEES** table for those employees whose **EMPLOYEE_ID** is greater than 200.
- Insert these values into the **SAL_HISTORY** and **MGR_HISTORY** tables using a multitable **INSERT**.

```
INSERT ALL
INTO sal_history VALUES(EMPID,HIREDATE,SAL)
INTO mgr_history VALUES(EMPID,MGR,SAL)

SELECT employee_id EMPID ,hire_date HIREDATE ,
       salary SAL , manager_id MGR
FROM   employees
WHERE  employee_id > 200;
```

8 rows created.



Conditional INSERT ALL

- Select the **EMPLOYEE_ID**, **HIRE_DATE**, **SALARY** and **MANAGER_ID** values from the **EMPLOYEES** table for those employees whose **EMPLOYEE_ID** is greater than 200.
- If the **SALARY** is greater than \$10,000, insert these values into the **SAL_HISTORY** table using a conditional multitable **INSERT** statement.
- If the **MANAGER_ID** is greater than 200, insert these values into the **MGR_HISTORY** table using a conditional multitable **INSERT** statement.

Conditional INSERT ALL

```
INSERT ALL
WHEN SAL > 10000 THEN
INTO sal_history VALUES(EMPID,HIREDATE,SAL)
WHEN MGR > 200 THEN
INTO mgr_history VALUES(EMPID,MGR,SAL)
SELECT employee_id EMPID,hire_date HIREDATE ,
       salary SAL , manager_id MGR
FROM employees
WHERE employee_id > 200;
4 rows created.
```

4 rows created.



Conditional FIRST INSERT

- Select the DEPARTMENT_ID , SUM(SALARY) and MAX(HIRE_DATE) from the EMPLOYEES table.
- If the SUM(SALARY) is greater than \$25,000 then insert these values into the SPECIAL_SAL, using a conditional FIRST multitable INSERT.
- If the first WHEN clause evaluates to true, the subsequent WHEN clauses for this row should be skipped.
- For the rows that do not satisfy the first WHEN condition, insert into the HIREDATE_HISTORY_00, or HIREDATE_HISTORY_99, or HIREDATE_HISTORY tables, based on the value in the HIRE_DATE column using a conditional multitable INSERT.

Conditional FIRST INSERT

```
INSERT FIRST
WHEN SAL > 25000 THEN
INTO special_sal VALUES(DEPTID, SAL)
WHEN HIREDATE like ('%00%') THEN
INTO hiredate_history_00 VALUES(DEPTID,HIREDATE)
WHEN HIREDATE like ('%99%') THEN
INTO hiredate_history_99 VALUES(DEPTID, HIREDATE)
ELSE
INTO hiredate_history VALUES(DEPTID, HIREDATE)
SELECT department_id DEPTID, SUM(salary) SAL,
       MAX(hire_date) HIREDATE
FROM employees
GROUP BY department_id;
```

8 rows created.

Pivoting INSERT

- Suppose you receive a set of sales records from a nonrelational database table, **SALES_SOURCE_DATA** in the following format:
EMPLOYEE_ID, WEEK_ID, SALES_MON,
SALES_TUE, SALES_WED, SALES_THUR,
SALES_FRI
- You would want to store these records in the **SALES_INFO** table in a more typical relational format:
EMPLOYEE_ID, WEEK, SALES
- Using a **pivoting** INSERT, convert the set of sales records from the nonrelational database table to relational format.

Pivoting INSERT

```
INSERT ALL
INTO sales_info VALUES (employee_id,week_id,sales_MON)
INTO sales_info VALUES (employee_id,week_id,sales_TUE)
INTO sales_info VALUES (employee_id,week_id,sales_WED)
INTO sales_info VALUES (employee_id,week_id,sales_THUR)
INTO sales_info VALUES (employee_id,week_id, sales_FRI)
SELECT EMPLOYEE_ID, week_id, sales_MON, sales_TUE,
       sales_WED, sales_THUR,sales_FRI
FROM sales_source_data;
```

5 rows created.

External Tables

- External tables are read-only tables in which the data is stored outside the database in flat files.
- The metadata for an external table is created using a CREATE TABLE statement.
- With the help of external tables, Oracle data can be stored or unloaded as flat files.
- The data can be queried using SQL but you cannot use DML and no indexes can be created.

Creating an External Table

- Use the `external_table_clause` along with the `CREATE TABLE` syntax to create an external table.
- Specify `ORGANIZATION AS EXTERNAL` to indicate that the table is located outside the database.
- The `external_table_clause` consists of the access driver `TYPE`, `external_data_properties`, and the `REJECT LIMIT`.
- The `external_data_properties` consist of the following:
 - `DEFAULT DIRECTORY`
 - `ACCESS PARAMETERS`
 - `LOCATION`

Example of Creating an External Table

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

```
CREATE DIRECTORY emp_dir AS '/flat_files' ;
```



Example of Creating an External Table

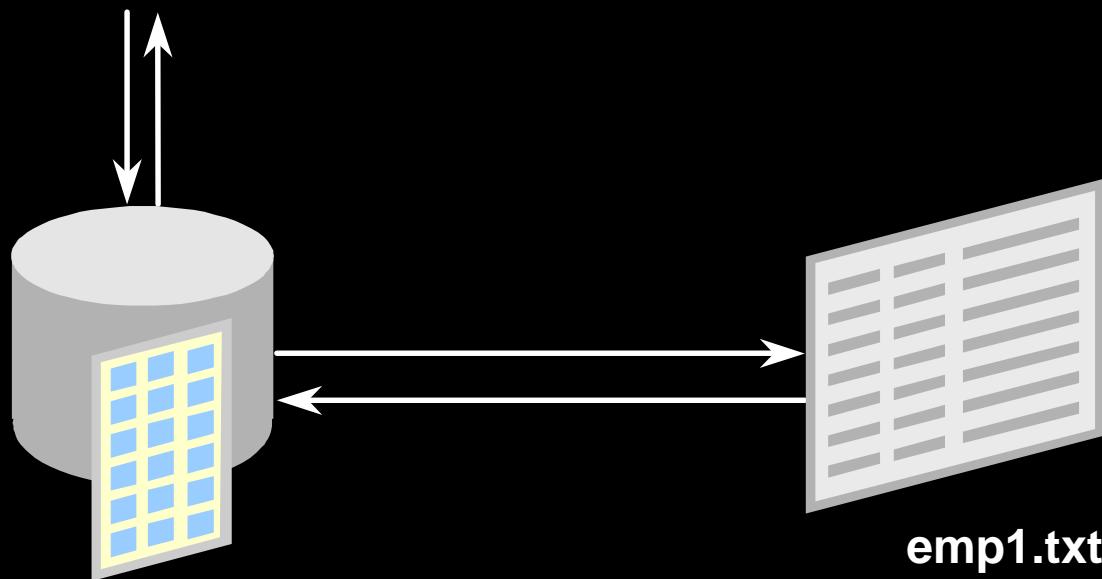
```
CREATE TABLE oldemp (
    empno NUMBER, empname CHAR(20), birthdate DATE)
ORGANIZATION EXTERNAL
(TYPE ORACLE_LOADER
DEFAULT DIRECTORY emp_dir
ACCESS PARAMETERS
(RECORDS DELIMITED BY NEWLINE
BADFILE 'bad_emp'
LOGFILE 'log_emp'
FIELDS TERMINATED BY ','
(empno CHAR,
empname CHAR,
birthdate CHAR date_format date mask "dd-mon-yyyy"))
LOCATION ('empl.txt'))
PARALLEL 5
REJECT LIMIT 200;
```

Table created.



Querying External Tables

```
SELECT *  
FROM oldemp
```



CREATE INDEX with CREATE TABLE Statement

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

Table created.

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

INDEX_NAME	TABLE_NAME
EMP_ID_IDX	NEW_EMP

Summary

In this lesson, you should have learned how to use the following enhancements to DML and DDL statements:

- **The INSERT...SELECT statement can be used to insert rows into multiple tables as part of a single DML statement.**
- **External tables can be created.**
- **Indexes can be named using the CREATE INDEX statement along with the CREATE TABLE statement.**

Practice 20 Overview

This practice covers the following topics:

- **Writing unconditional INSERT**
- **Writing conditional ALL INSERT**
- **Pivoting INSERT**
- **Creating indexes along with the CREATE TABLE command**



Using SQL*Plus

ORACLE®

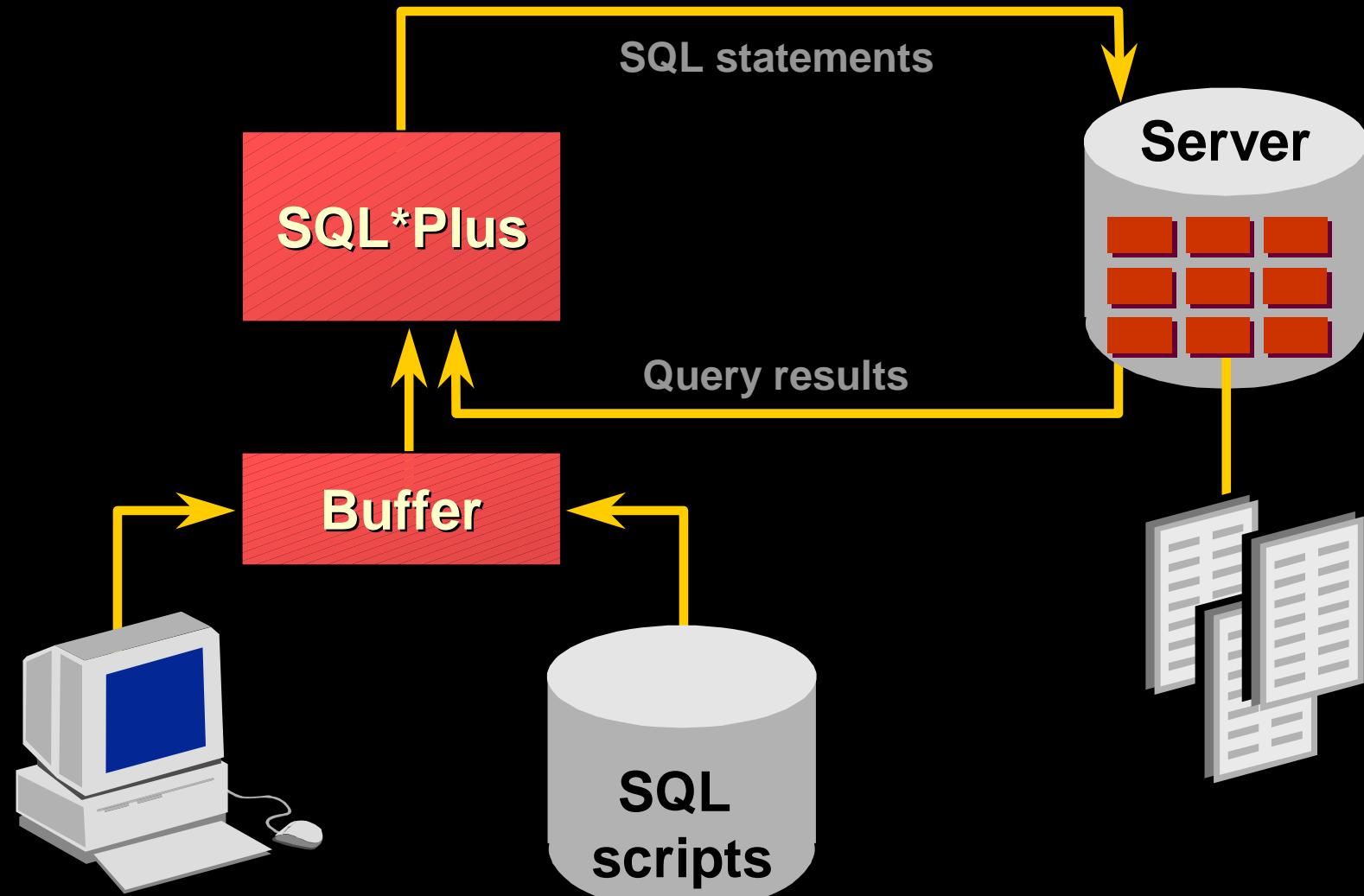
Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

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

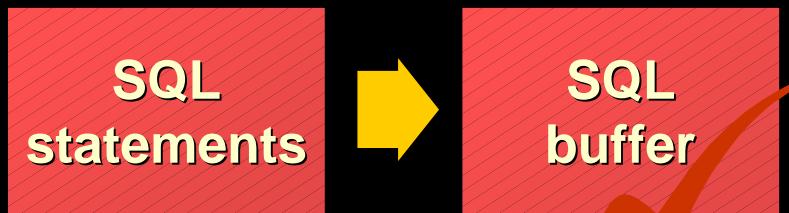
SQL and SQL*Plus Interaction



SQL Statements versus SQL*Plus Commands

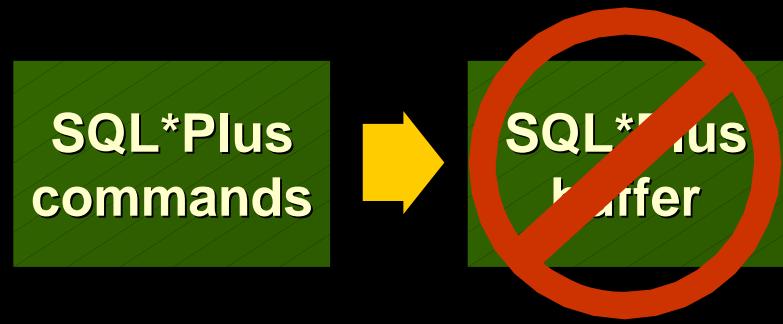
SQL

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



SQL*Plus

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

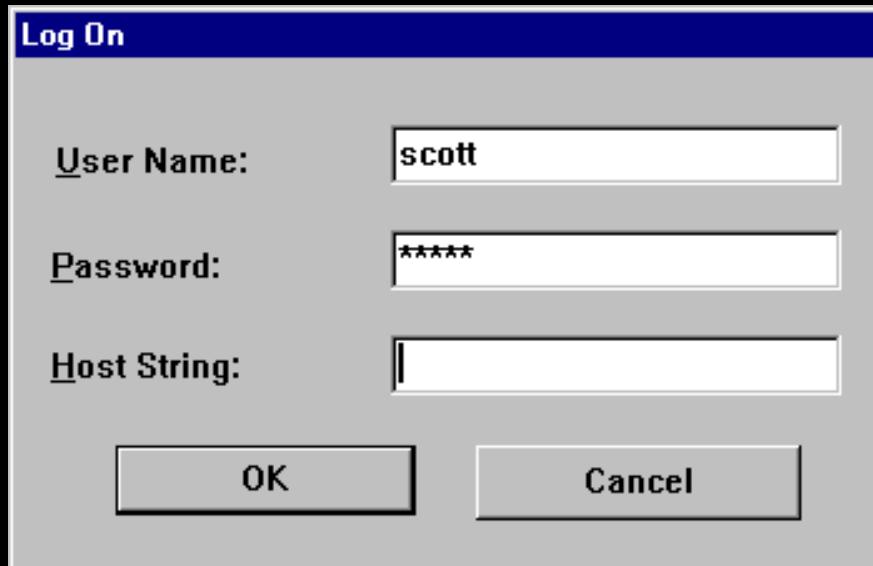


Overview of SQL*Plus

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

Logging In to SQL*Plus

- 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[PEND]** *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***
- **O *text***

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

Use SQL*Plus as an environment to:

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

Writing Advanced Scripts

ORACLE®

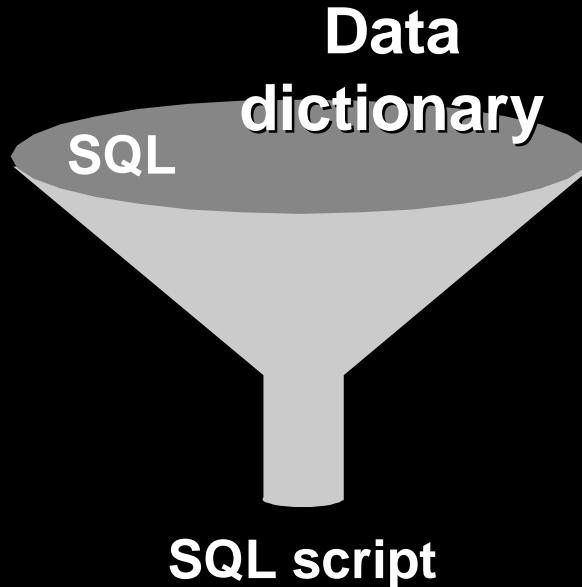
Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Describe the types of problems that are solved by using SQL to generate SQL**
- **Write a script that generates a script of `DROP TABLE` statements**
- **Write a script that generates a script of `INSERT INTO` statements**

Using SQL to Generate SQL



- **SQL can be used to generate scripts in SQL**
- **The data dictionary**
 - **Is a collection of tables and views that contain database information**
 - **Is created and maintained by the Oracle server**

Creating a Basic Script

```
SELECT 'CREATE TABLE ' || table_name || '_test '
      || 'AS SELECT * FROM ' || table_name
      || ' WHERE 1=2;'
      AS "Create Table Script"
FROM user_tables;
```

Create Table Script

```
CREATE TABLE COUNTRIES_test AS SELECT * FROM COUNTRIES WHERE 1=2;
CREATE TABLE DEPARTMENTS_test AS SELECT * FROM DEPARTMENTS WHERE 1=2;
CREATE TABLE EMPLOYEES_test AS SELECT * FROM EMPLOYEES WHERE 1=2;
CREATE TABLE JOBS_test AS SELECT * FROM JOBS WHERE 1=2;
CREATE TABLE JOB_GRADES_test AS SELECT * FROM JOB_GRADES WHERE 1=2;
CREATE TABLE JOB_HISTORY_test AS SELECT * FROM JOB_HISTORY WHERE 1=2;
CREATE TABLE LOCATIONS_test AS SELECT * FROM LOCATIONS WHERE 1=2;
CREATE TABLE REGIONS_test AS SELECT * FROM REGIONS WHERE 1=2;
```

8 rows selected.

ORACLE

Controlling the Environment

```
SET ECHO OFF  
SET FEEDBACK OFF  
SET PAGESIZE 0
```

Set system variables to appropriate values.

```
SPOOL dropem.sql
```

SQL STATEMENT

```
SPOOL OFF
```

```
SET FEEDBACK ON  
SET PAGESIZE 24  
SET ECHO ON
```

Set system variables back to the default value.

The Complete Picture

```
SET ECHO OFF
SET FEEDBACK OFF
SET PAGESIZE 0

SELECT 'DROP TABLE ' || object_name || ';' 
FROM user_objects
WHERE object_type = 'TABLE'
/

SET FEEDBACK ON
SET PAGESIZE 24
SET ECHO ON
```

Dumping the Contents of a Table to a File

```
SET HEADING OFF ECHO OFF FEEDBACK OFF
SET PAGESIZE 0

SELECT
  'INSERT INTO departments_test VALUES
  (' || department_id || ', ''' || department_name || ''
  '', '' || location_id || ''');'
  AS "Insert Statements Script"
FROM   departments
/
SET PAGESIZE 24
SET HEADING ON ECHO ON FEEDBACK ON
```

Dumping the Contents of a Table to a File

Source	Result
' ''X'' '	'X'
' .. '	'
' '' ' department_name ' '' '	'Administration'
' '' , '' '	','
' '') ; '	') ;

Generating a Dynamic Predicate

```
COLUMN my_col NEW_VALUE dyn_where_clause

SELECT DECODE('&&deptno', null,
DECODE ('&&hiredate', null, ' ',
'WHERE hire_date=TO_DATE('' || '&&hiredate'', ''DD-MON-YYYY'' )'),
DECODE ('&&hiredate', null,
'WHERE department_id = ' || '&&deptno',
'WHERE department_id = ' || '&&deptno' ||
' AND hire_date = TO_DATE('' || '&&hiredate'', ''DD-MON-YYYY'' )')
AS my_col FROM dual;

SELECT last_name FROM employees &dyn_where_clause;
```



Summary

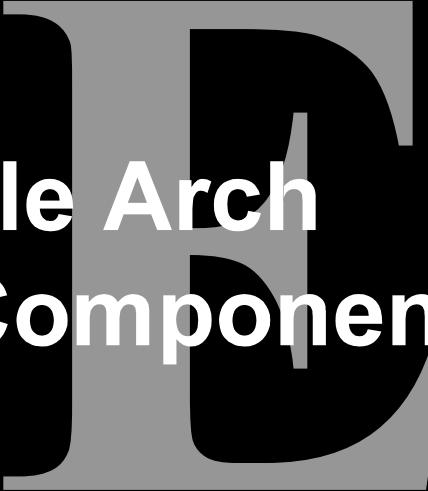
In this appendix, you should have learned the following:

- You can write a SQL script to generate another SQL script.
- Script files often use the data dictionary.
- You can capture the output in a file.

Practice D Overview

This practice covers the following topics:

- **Writing a script to describe and select the data from your tables**
- **Writing a script to revoke user privileges**



Oracle Arch Components

ORACLE®

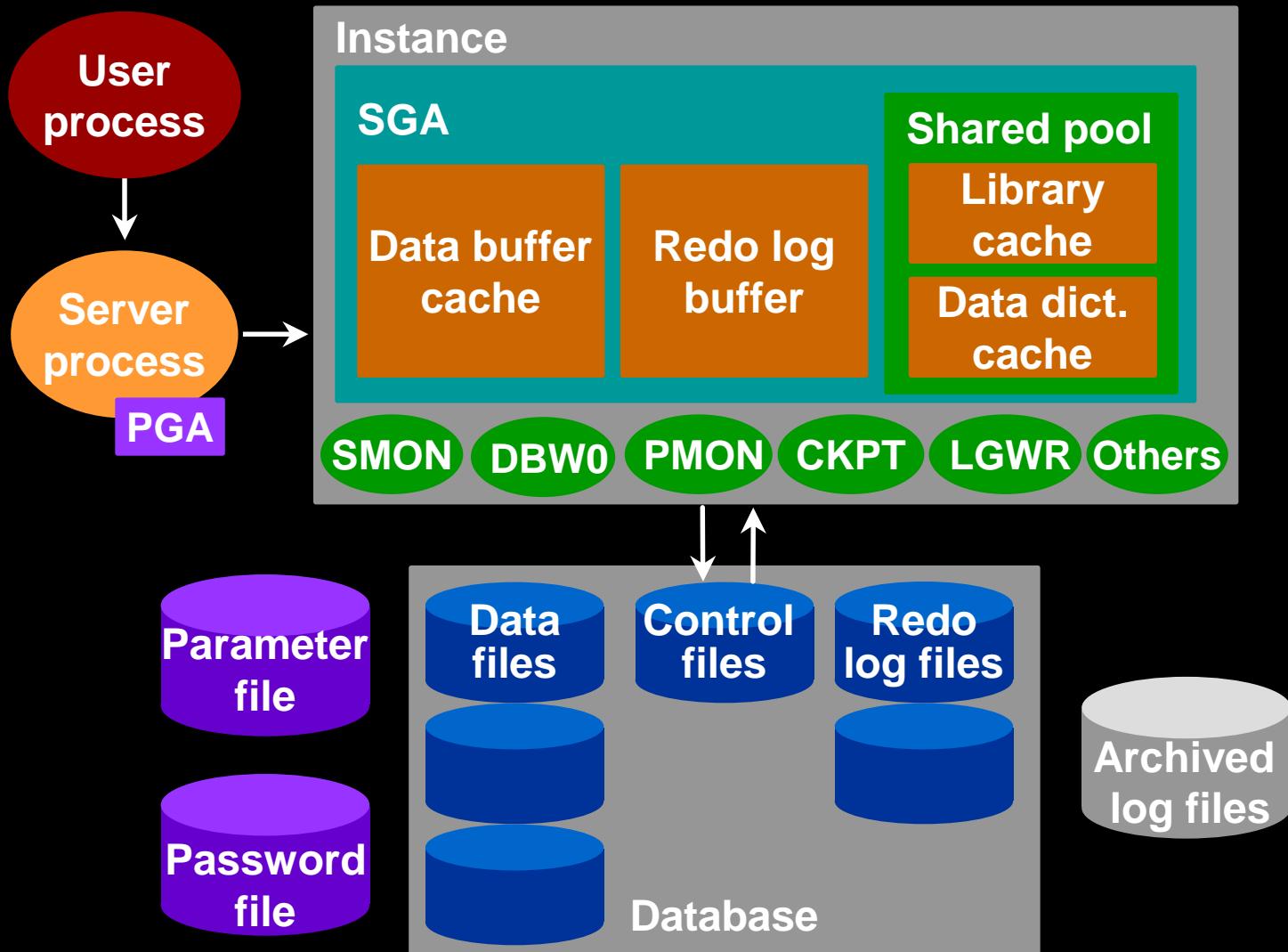
Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

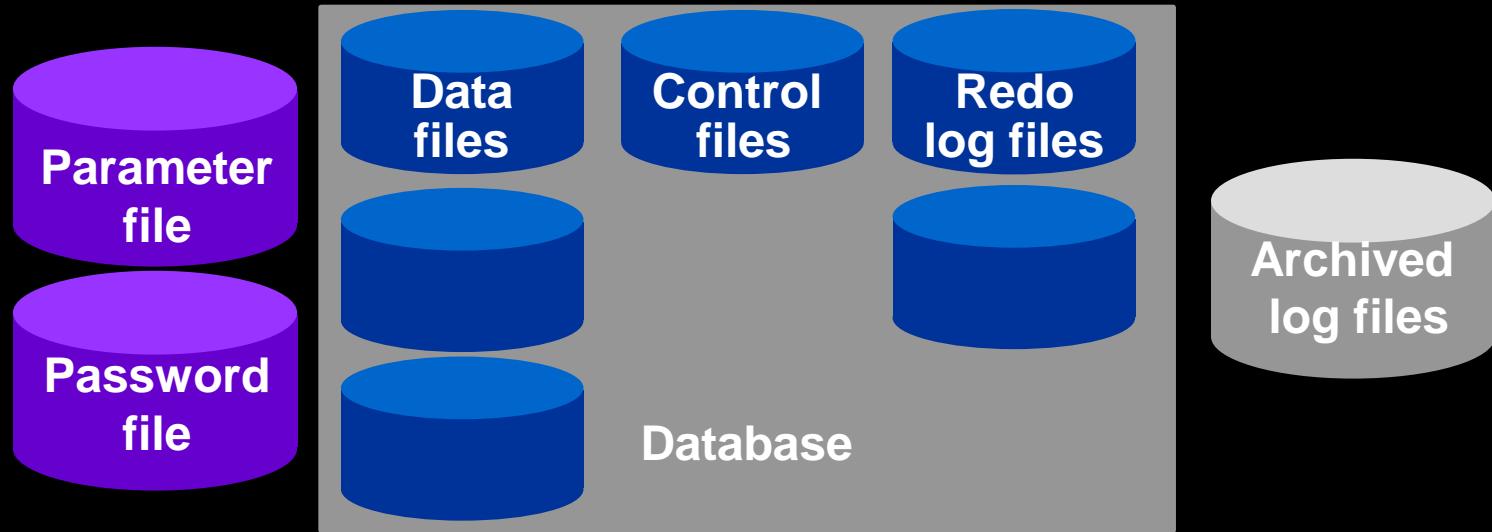
- **Describe the Oracle Server architecture and its main components**
- **List the structures involved in connecting a user to an Oracle instance**
- **List the stages in processing:**
 - **Queries**
 - **DML statements**
 - **Commits**

Overview

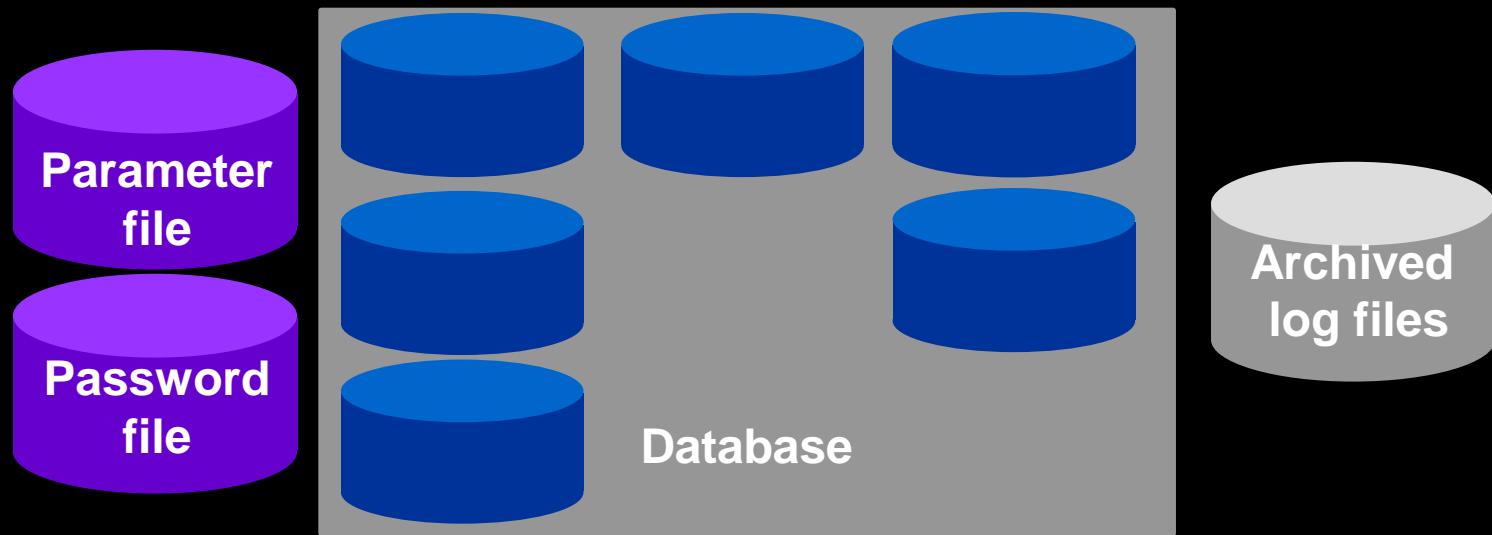


ORACLE®

Oracle Database Files



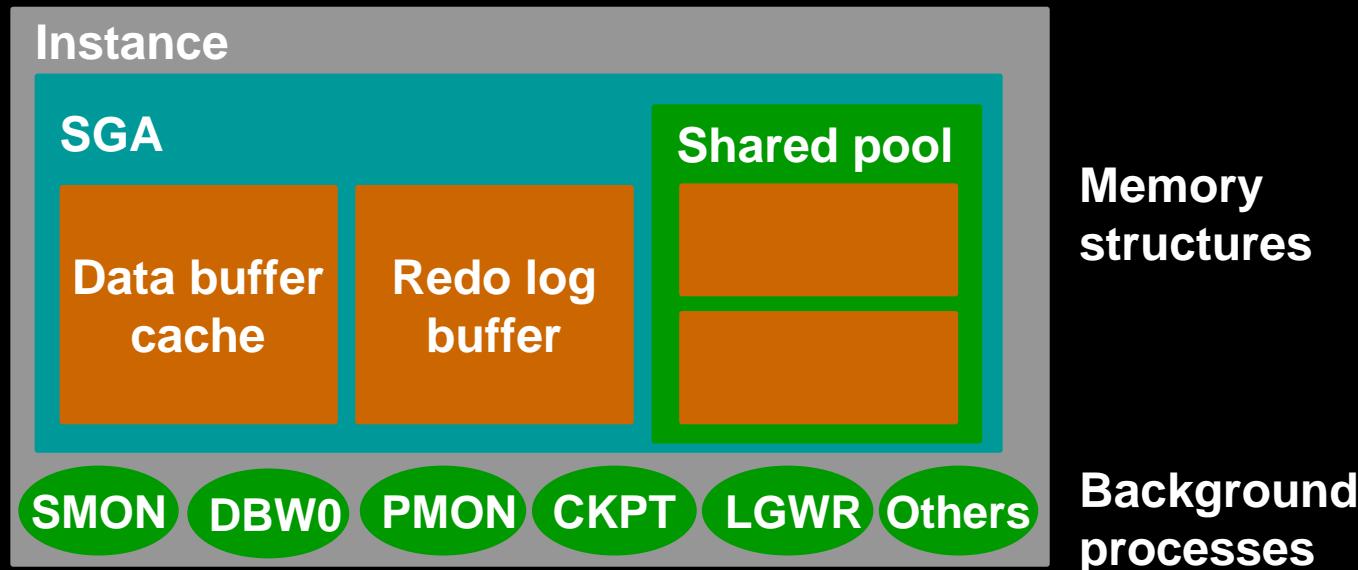
Other Key Physical Structures



Oracle Instance

An Oracle instance:

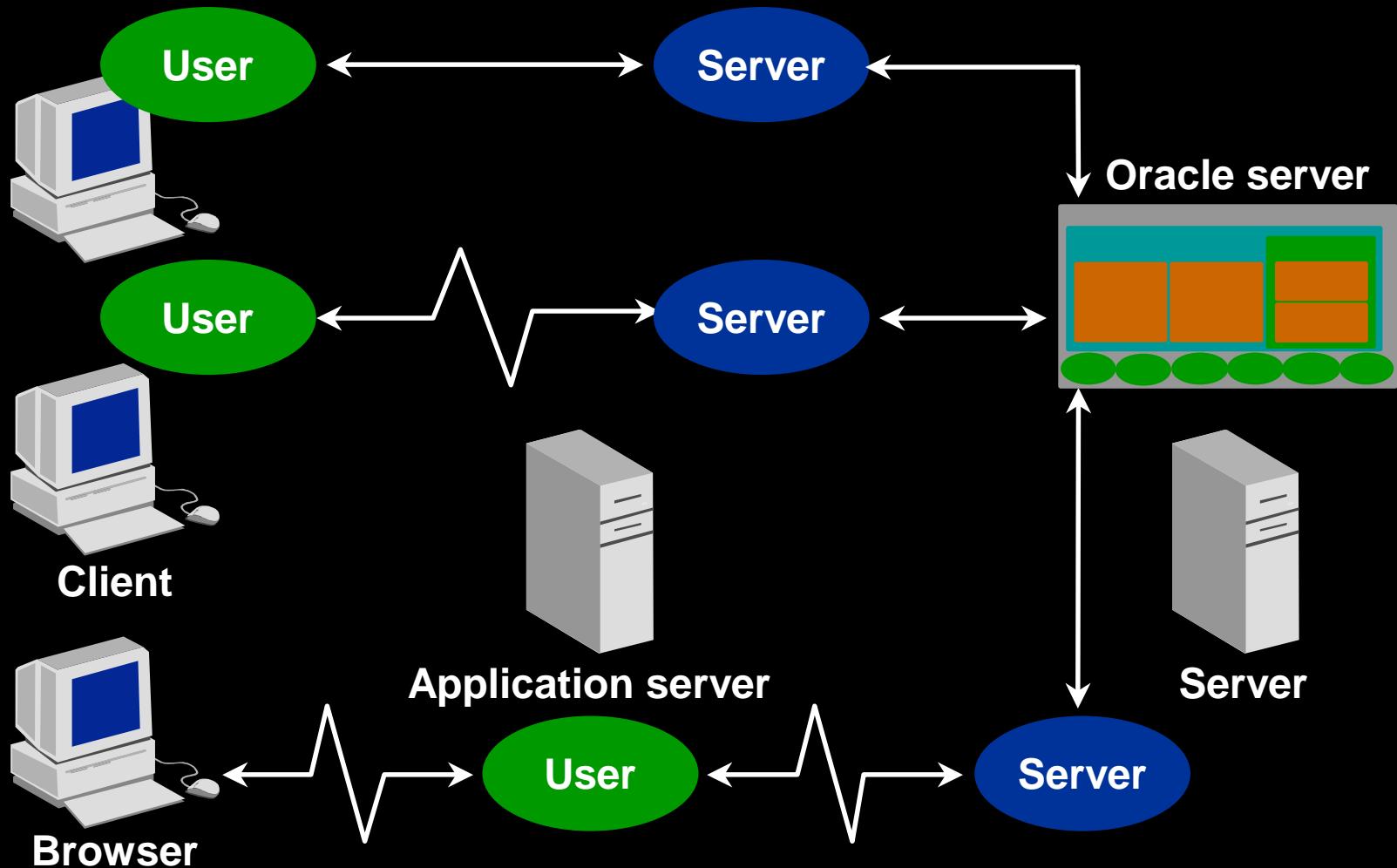
- Is a means to access an Oracle database
- Always opens one and only one database



Processing a SQL Statement

- **Connect to an instance using:**
 - The user process
 - The server process
- **The Oracle Server components that are used depend on the type of SQL statement:**
 - Queries return rows
 - DML statements log changes
 - Commit ensures transaction recovery
- **Some Oracle Server components do not participate in SQL statement processing.**

Connecting to an Instance

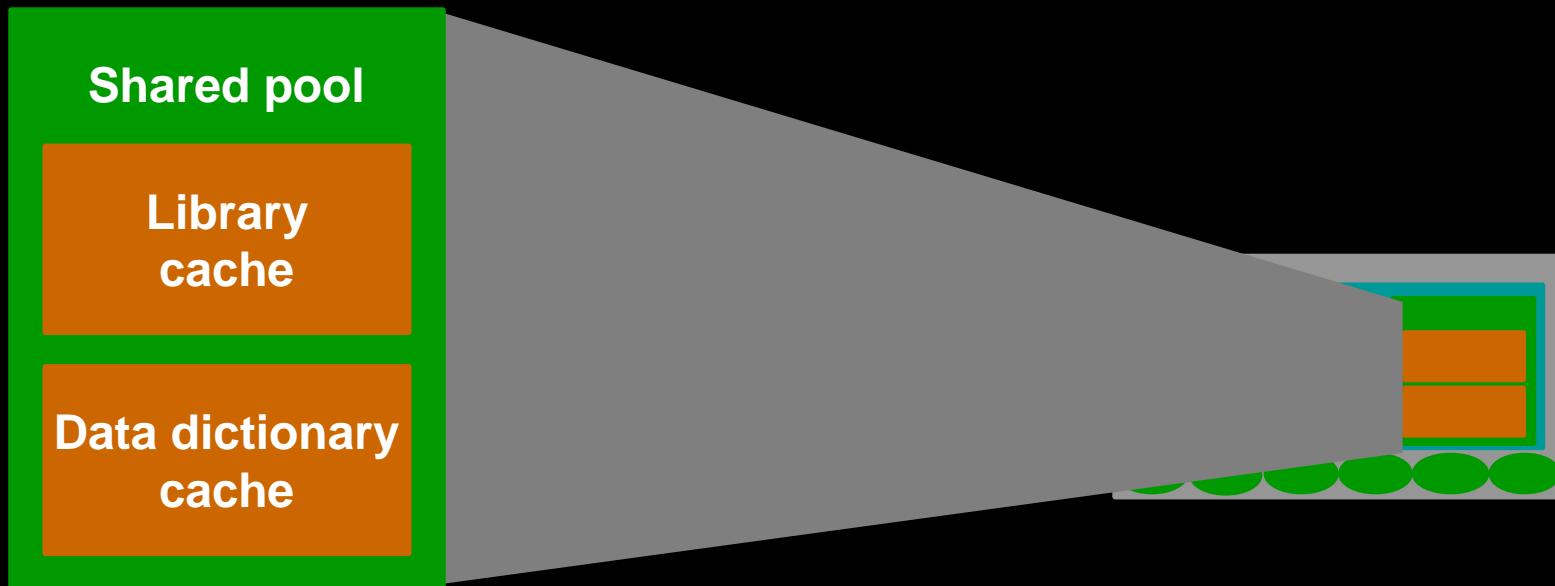


ORACLE

Processing a Query

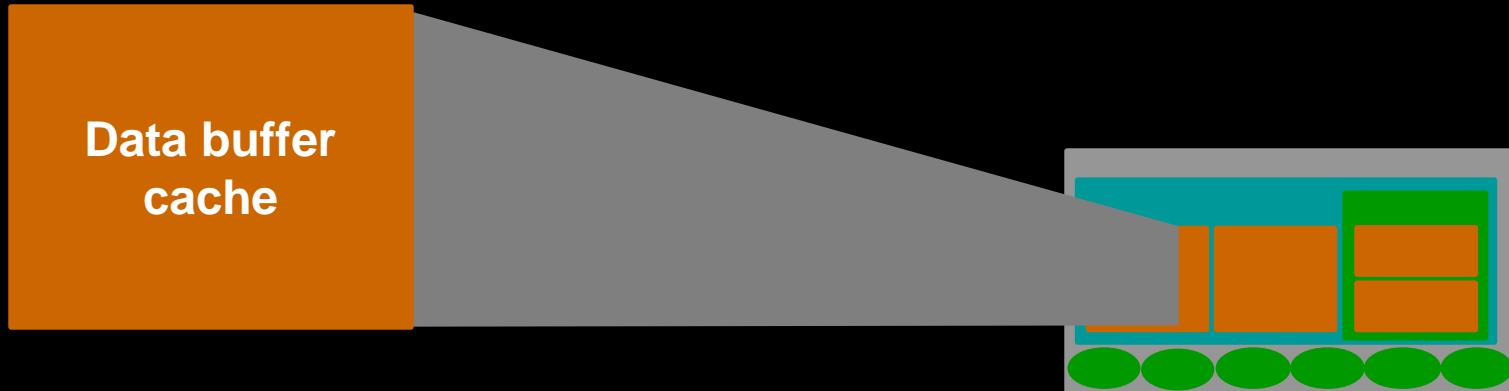
- **Parse:**
 - Search for identical statement
 - Check syntax, object names, and privileges
 - Lock objects used during parse
 - Create and store execution plan
- **Execute: Identify rows selected**
- **Fetch: Return rows to user process**

The Shared Pool



- The library cache contains the SQL statement text, parsed code, and execution plan.
- The data dictionary cache contains table, column, and other object definitions and privileges.
- The shared pool is sized by SHARED_POOL_SIZE.

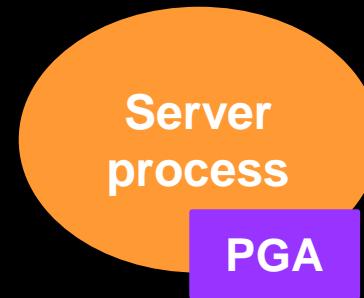
Database Buffer Cache



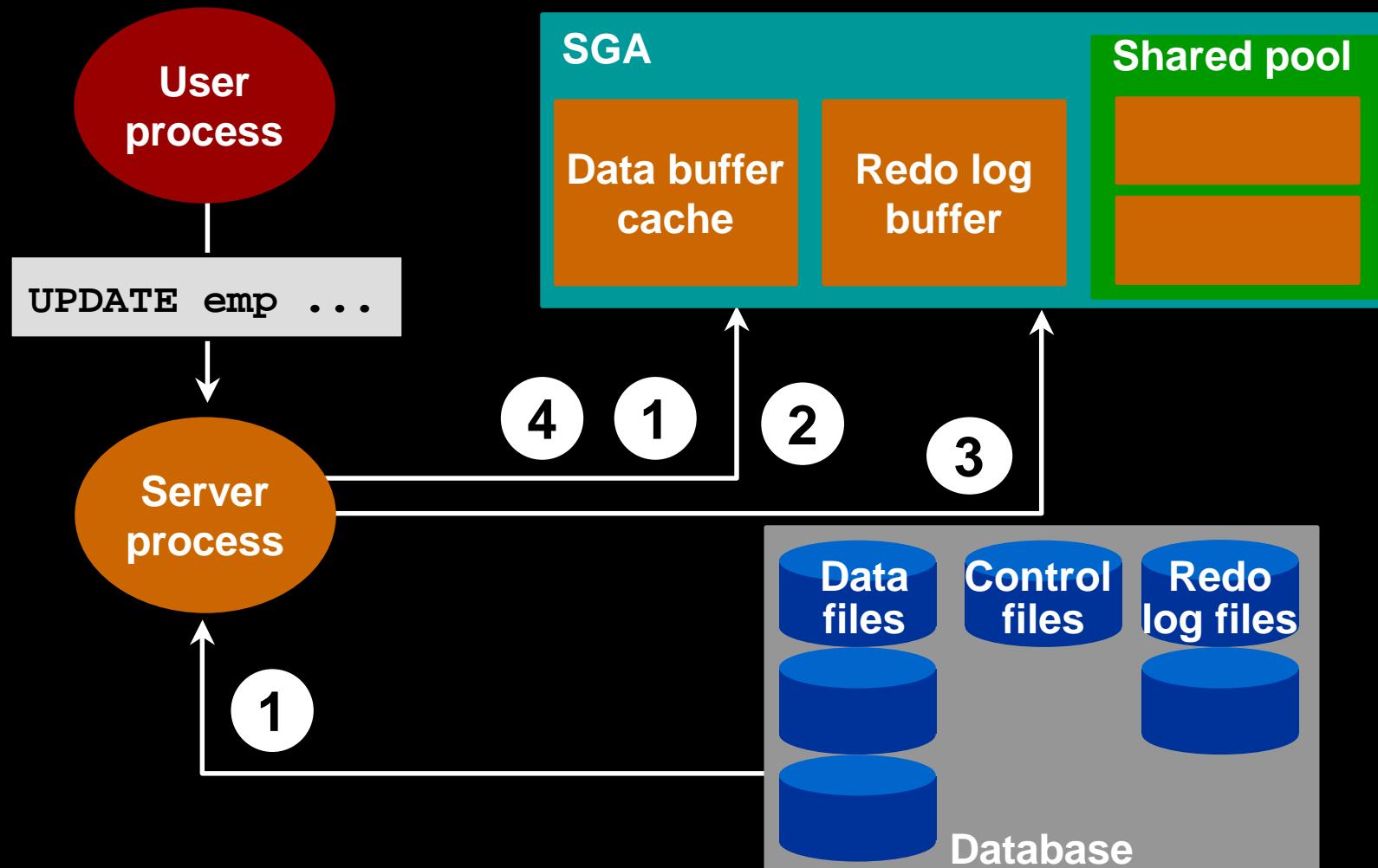
- **Stores the most recently used blocks**
- **Size of a buffer based on DB_BLOCK_SIZE**
- **Number of buffers defined by DB_BLOCK_BUFFERS**

Program Global Area (PGA)

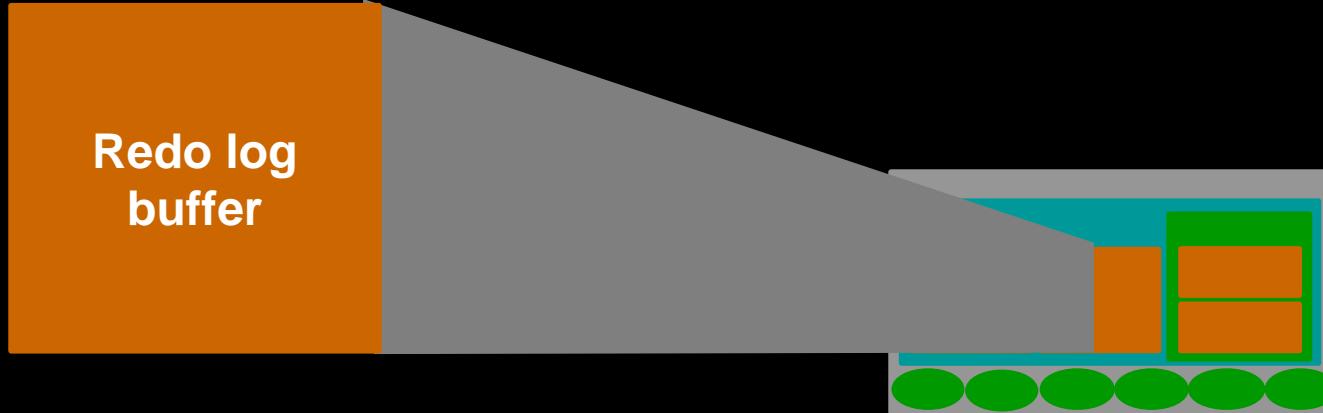
- Not shared
- Writable only by the server process
- Contains:
 - Sort area
 - Session information
 - Cursor state
 - Stack space



Processing a DML Statement

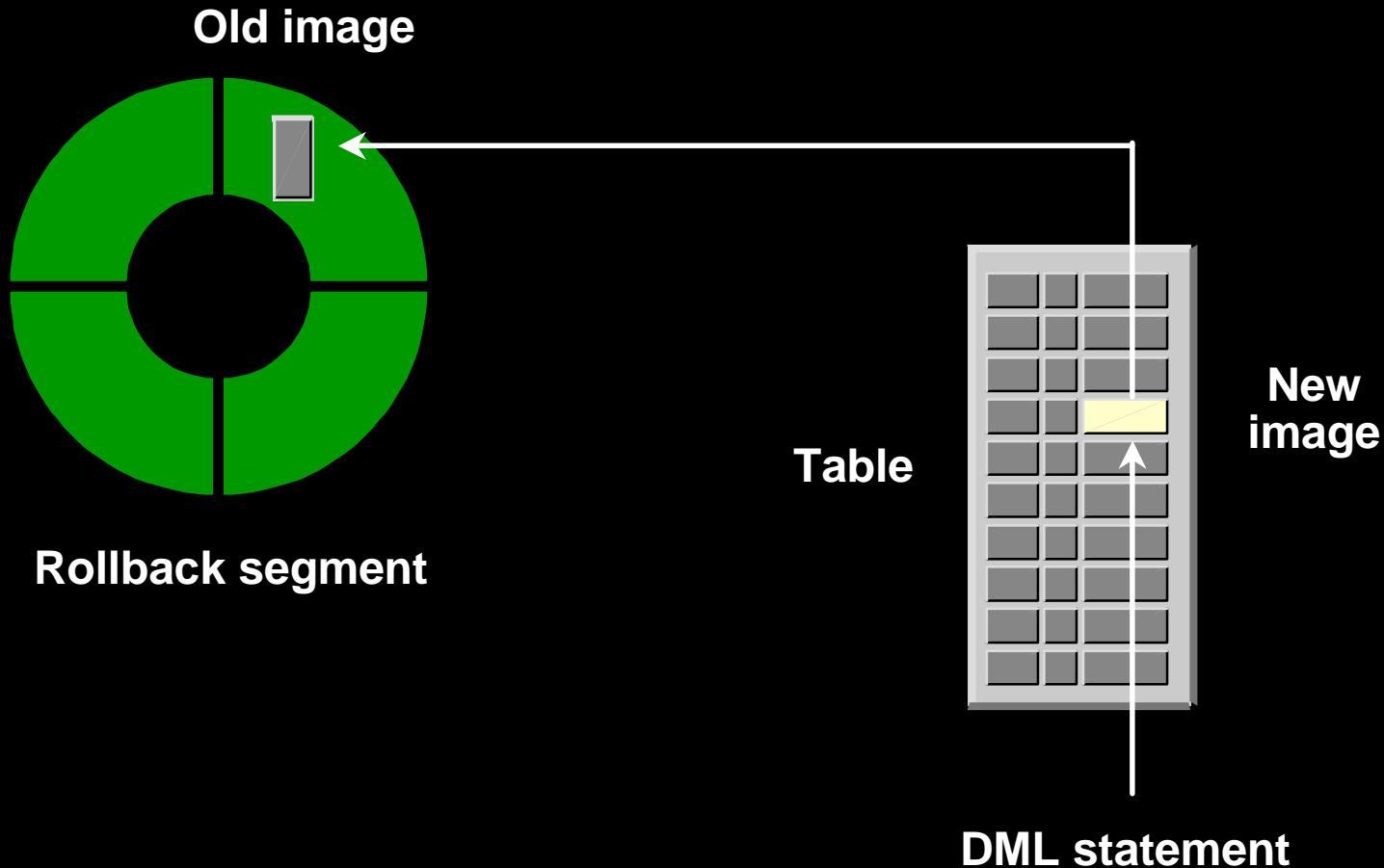


Redo Log Buffer

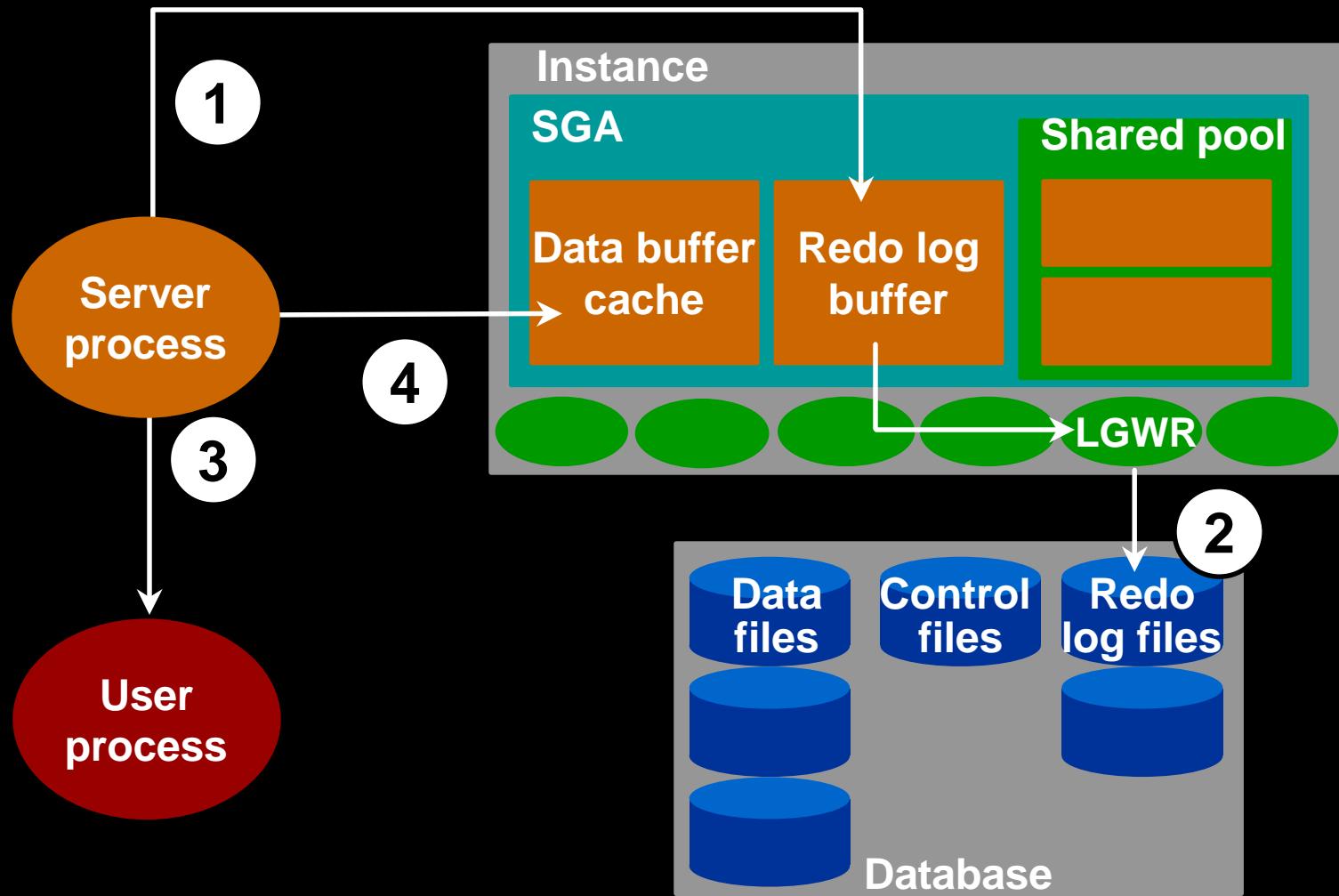


- Has its size defined by `LOG_BUFFER`
- Records changes made through the instance
- Is used sequentially
- Is a circular buffer

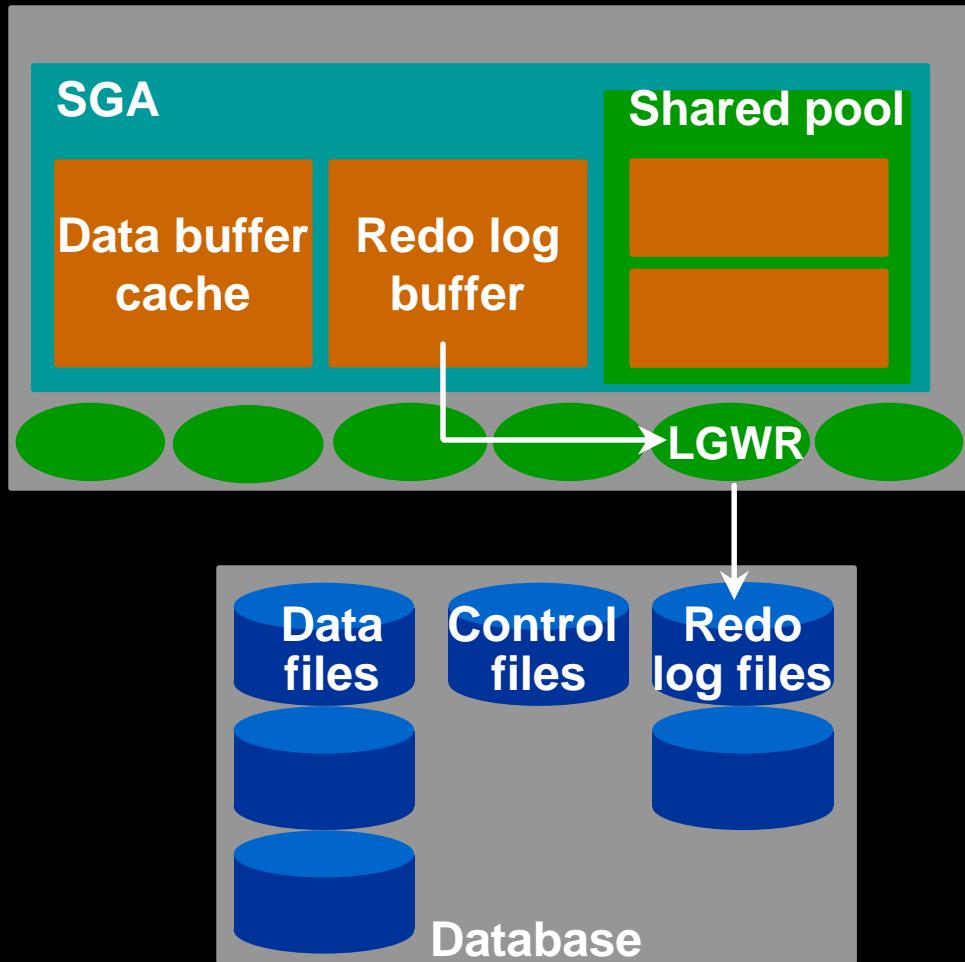
Rollback Segment



COMMIT Processing



Log Writer (LGWR)



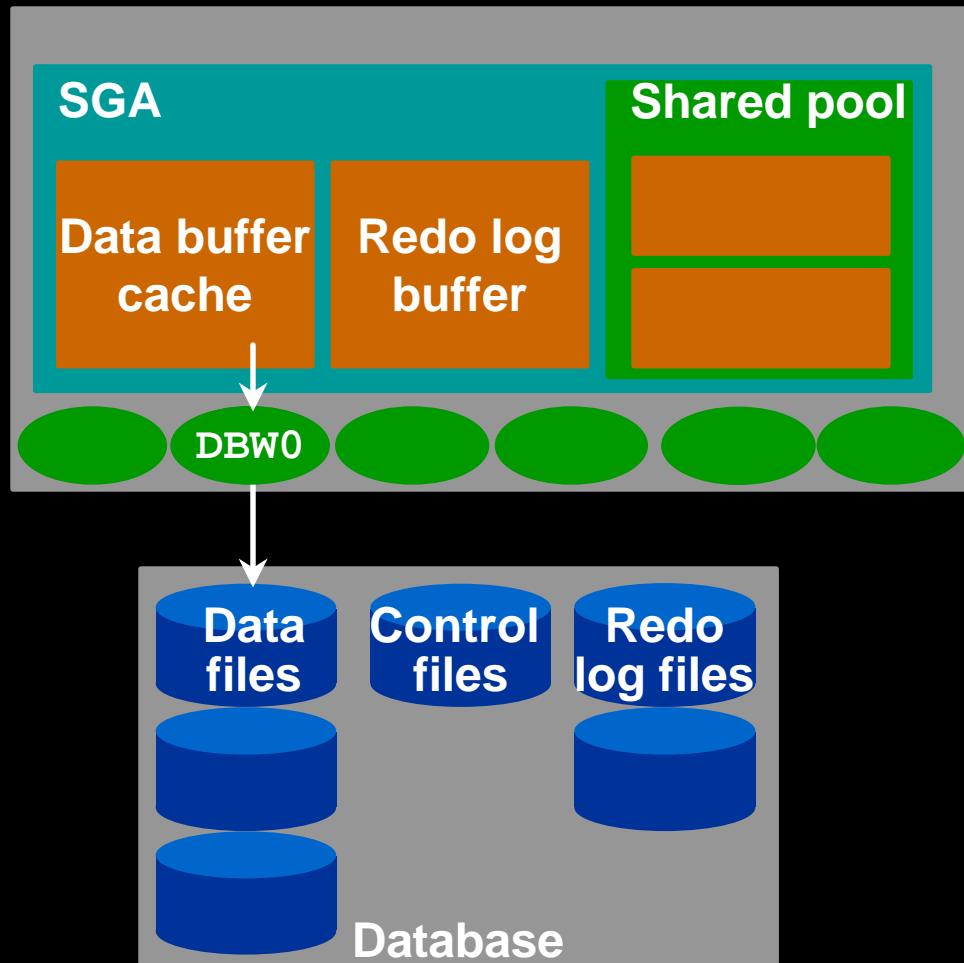
LGWR writes when:

- There is a commit
- The redo buffer log is one-third full
- There is more than 1 MB of redo
- Before DBW0 writes

Other Instance Processes

- **Other required processes:**
 - Database Writer (DBW0)
 - Process Monitor (PMON)
 - System Monitor (SMON)
 - Checkpoint (CKPT)
- **The archive process (ARC0) is usually created in a production database**

Database Writer (DBW0)



DBW0 writes when:

- There are many dirty buffers
- There are few free buffers
- Timeout occurs
- Checkpoint occurs

SMON: System Monitor

- **Automatically recovers the instance:**
 - Rolls forward changes in the redo logs
 - Opens the database for user access
 - Rolls back uncommitted transactions
- **Coalesces free space**
- **Deallocates temporary segments**

PMON: Process Monitor

Cleans up after failed processes by:

- **Rolling back the transaction**
- **Releasing locks**
- **Releasing other resources**

Summary

In this appendix, you should have learned how to:

- **Identify database files: data files, control files, online redo logs**
- **Describe SGA memory structures: DB buffer cache, shared SQL pool, and redo log buffer**
- **Explain primary background processes: DBW0, LGWR, CKPT, PMON, SMON, and ARC0**
- **List SQL processing steps: parse, execute, fetch**