
Oracle9i: Advanced SQL

Student Guide • Volume 1

40058GC11
Production 1.1
November 2001
D34074

ORACLE®

Authors

Priya Nathan

Technical Contributors and Reviewers

Josephine Turner
Martin Alvarez
Anna Atkinson
Don Bates
Marco Berbeek
Andrew Brannigan
Laszlo Czinkoczki
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.

Contents

I Introduction

Course Objectives 1-2

Course Overview 1-3

1 Using SET Operators

Objectives 1-2

The SET Operators 1-3

Tables Used in This Lesson 1-4

The UNION Operator 1-7

Using the UNION Operator 1-8

The UNION ALL Operator 1-10

Using the UNION ALL Operator 1-11

The INTERSECT Operator 1-12

Using the INTERSECT Operator 1-13

The MINUS Operator 1-14

SET Operator Guidelines 1-16

The Oracle Server and SET Operators 1-17

Matching the SELECT Statements 1-18

Controlling the Order of Rows 1-20

Summary 1-21

Practice 1 Overview 1-22

2 Oracle9i Datetime Functions

Objectives 2-2

TIME ZONES 2-3

Oracle9i Datetime Support 2-4

TZ_OFFSET 2-6

CURRENT_DATE 2-8

CURRENT_TIMESTAMP 2-9

LOCALTIMESTAMP 2-10

DBTIMEZONE and SESSIONTIMEZONE 2-11

EXTRACT 2-12

TIMESTAMP Conversion Using FROM_TZ 2-13

STRING To TIMESTAMP Conversion Using TO_TIMESTAMP and
TO_TIMESTAMP_TZ 2-14

Time Interval Conversion with TO_YMINTERVAL 2-15

Summary 2-16

Practice 2 Overview 2-17

3 Enhancements to the GROUP BY Clause

- Objectives 3-2
- Review of Group Functions 3-3
- Review of the GROUP BY Clause 3-4
- Review of the HAVING Clause 3-5
- GROUP BY with ROLLUP and CUBE Operators 3-6
- ROLLUP Operator 3-7
- ROLLUP Operator Example 3-8
- CUBE Operator 3-9
- CUBE Operator: Example 3-10
- GROUPING Function 3-11
- GROUPING Function: Example 3-12
- GROUPING SETS 3-13
- GROUPING SETS: Example 3-15
- Composite Columns 3-17
- Composite Columns: Example 3-19
- Concatenated Groupings 3-21
- Concatenated Groupings Example 3-22
- Summary 3-23
- Practice 3 Overview 3-24

4 Advanced Subqueries

- Objectives 4-2
- What Is a Subquery? 4-3
- Subqueries 4-4
- Using a Subquery 4-5
- Multiple-Column Subqueries 4-6
- Column Comparisons 4-7
- Pairwise Comparison Subquery 4-8
- Nonpairwise Comparison Subquery 4-9
- Using a Subquery in the FROM Clause 4-10
- Scalar Subquery Expressions 4-11
- Scalar Subqueries: Examples 4-12
- Correlated Subqueries 4-14
- Using Correlated Subqueries 4-16
- Using the EXISTS Operator 4-18
- Using the NOT EXISTS Operator 4-20
- Correlated UPDATE 4-21
- The WITH Clause 4-26
- WITH Clause: Example 4-27
- Summary 4-29
- Practice 4 Overview 4-31

5 Hierarchical Retrieval

- Objectives 5-2
- Sample Data from the EMPLOYEES Table 5-3
- Natural Tree Structure 5-4
- Hierarchical Queries 5-5
- Walking the Tree 5-6
- Walking the Tree: From the Bottom Up 5-8
- Walking the Tree: From the Top Down 5-9
- Ranking Rows with the LEVEL Pseudocolumn 5-10
- Formatting Hierarchical Reports Using LEVEL and LPAD 5-11
- Pruning Branches 5-13
- Summary 5-14
- Practice 5 Overview 5-15

6 Oracle9i Extensions to DML and DDL Statements

- Objectives 6-2
- Review of the INSERT Statement 6-3
- Review of the UPDATE Statement 6-4
- Overview of Multitable INSERT Statements 6-5
- Types of Multitable INSERT Statements 6-7
- Multitable INSERT Statements 6-8
- Unconditional INSERT ALL 6-10
- Conditional INSERT ALL 6-11
- Conditional FIRST INSERT 6-13
- Pivoting INSERT 6-15
- External Tables 6-18
- Creating an External Table 6-19
- Example of Creating an External Table 6-20
- Querying External Tables 6-23
- CREATE INDEX with CREATE TABLE Statement 6-24
- Summary 6-25
- Practice 6 Overview 6-26

A Practice Solutions

B Table Descriptions and Data

C Writing Advanced Scripts

D Oracle Architectural Components

Index

Additional Practices

Additional Practice Solutions

Additional Practices: Table Descriptions and Data

Preface

Profile

Before You Begin This Course

Before you begin this course, you should be able to use a graphical user interface (GUI). Required prerequisites are familiarity with data processing concepts and techniques.

How This Course Is Organized

Oracle9i: Advanced SQL is an instructor-led course featuring lectures and hands-on exercises. Online demonstrations and written practice sessions reinforce the concepts and skills introduced.

Related Publications

Oracle Publications

Title	Part Number
<i>Oracle9i Reference, Release 1 (9.0.1)</i>	A90190-02
<i>Oracle9i SQL Reference, Release 1 (9.0.1)</i>	A90125-01
<i>Oracle9i Concepts, Release 1 (9.0.0)</i>	A88856-02
<i>Oracle9i Server Application Developer's Guide Fundamentals Release 1 (9.0.1)</i>	A88876-02
<i>iSQL*Plus User's Guide and Reference, Release 9.0.0</i>	A88826-01
<i>SQL*Plus User's Guide and Reference, Release 9.0.1</i>	A88827-02

Additional Publications

- System release bulletins
- Installation and user's guides
- *read.me* files
- International Oracle User's Group (IOUG) articles
- *Oracle Magazine*

Typographic Conventions

What follows are two lists of typographical conventions used specifically within text or within code.

Typographic Conventions Within Text

Convention	Object or Term	Example
Uppercase	Commands, functions, column names, table names, PL/SQL objects, schemas	Use the <code>SELECT</code> command to view information stored in the <code>LAST_NAME</code> column of the <code>EMPLOYEES</code> table.
Lowercase, italic	Filenames, syntax variables, usernames, passwords	where: <i>role</i> is the name of the role to be created.
Initial cap	Trigger and button names	Assign a When-Validate-Item trigger to the ORD block. Choose Cancel.
Italic	Books, names of courses and manuals, and emphasized words or phrases	For more information on the subject see <i>Oracle Server SQL Language Reference Manual</i> Do <i>not</i> save changes to the database.
Quotation marks	Lesson module titles referenced within a course	This subject is covered in Lesson 3, “Working with Objects.”

Typographic Conventions (continued)

Typographic Conventions Within Code

Convention	Object or Term	Example
Uppercase	Commands, functions	<code>SELECT employee_id FROM employees;</code>
Lowercase, italic	Syntax variables	<code>CREATE ROLE <i>role</i>;</code>
Initial cap	Forms triggers	<code>Form module: ORD Trigger level: S_ITEM.QUANTITY item Trigger name: When-Validate-Item . . .</code>
Lowercase	Column names, table names, filenames, PL/SQL objects	<code>. . . OG_ACTIVATE_LAYER (OG_GET_LAYER ('prod_pie_layer')) . . . SELECT last_name FROM employees;</code>
Bold	Text that must be entered by a user	<code>CREATE USER scott IDENTIFIED BY tiger;</code>

I

Introduction

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Course Objectives

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

- **Use advanced SQL data retrieval techniques to retrieve data from database tables**
- **Apply advanced techniques in a practice that simulates real life**

ORACLE

Course Overview

Using advanced SQL data retrieval techniques such as:

- **SET operators**
- **Oracle9i DATETIME functions**
- **ROLLUP , CUBE operators and GROUPING SETS**
- **Hierarchical queries**
- **Correlated subqueries**
- **Multitable inserts**
- **External Tables**

ORACLE

1

Using SET Operators

ORACLE[®]

Copyright © Oracle Corporation, 2001. All rights reserved.

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**

ORACLE[®]

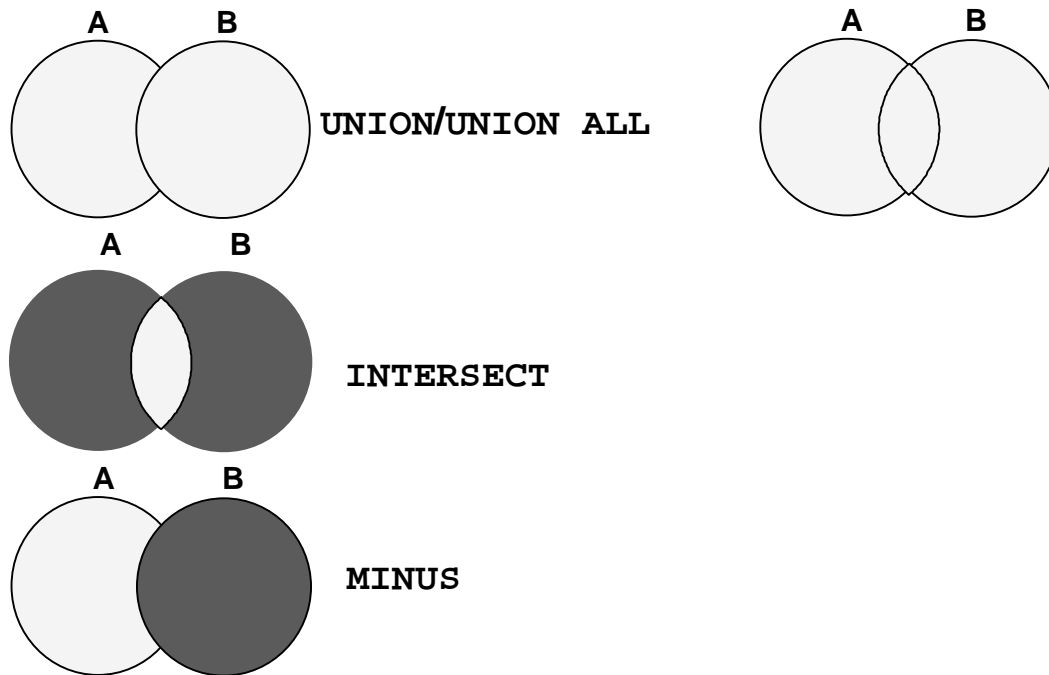
1-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

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

The SET Operators



ORACLE

1-3

Copyright © Oracle Corporation, 2001. All rights reserved.

The SET Operators

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

Operator	Returns
UNION	All distinct rows selected by either query
UNION ALL	All rows selected by either query, including all duplicates
INTERSECT	All distinct rows selected by both queries
MINUS	All distinct rows that are selected by the first SELECT statement and not selected in the second SELECT statement

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

Note: In the slide, the light color (gray) in the diagram represents the query result.

Tables Used in This Lesson

The tables used in this lesson are:

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

ORACLE

1-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Tables Used in This Lesson

Two tables are used in this lesson. They are the EMPLOYEES table and the JOB_HISTORY table.

The EMPLOYEES table stores the employee details. For the human resource records, this table stores a unique identification number and email address for each employee. The details of the employee's job identification number, salary, and manager are also stored. Some of the employees earn a commission in addition to their salary; this information is tracked too. The company organizes the roles of employees into jobs. Some of the employees have been with the company for a long time and have switched to different jobs. This is monitored using the JOB_HISTORY table. 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 the JOB_HISTORY table.

The structure and the data from the EMPLOYEES and the JOB_HISTORY tables are shown on the next page.

There have been instances in the company of people who have held the same position more than once during their tenure with the company. For example, consider the employee Taylor, who joined the company on 24-MAR-1998. Taylor held the job title SA_REP for the period 24-MAR-98 to 31-DEC-98 and the job title SA_MAN for the period 01-JAN-99 to 31-DEC-99. Taylor moved back into the job title of SA_REP, which is his current job title.

Similarly consider the employee Whalen, who joined the company on 17-SEP-1987. Whalen held the job title AD_ASST for the period 17-SEP-87 to 17-JUN-93 and the job title AC_ACCOUNT for the period 01-JUL-94 to 31-DEC-98. Whalen moved back into the job title of AD_ASST, which is his current job title.

Tables Used in This Lesson (continued)

DESC 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)
DEPARTMENT_NAME		VARCHAR2(14)

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

EMPLOYEE_ID	LAST_NAME	JOB_ID	HIRE_DATE	DEPARTMENT_ID
100	King	AD_PRES	17-JUN-87	90
101	Kochhar	AD_VP	21-SEP-89	90
102	De Haan	AD_VP	13-JAN-93	90
103	Hunold	IT_PROG	03-JAN-90	60
104	Ernst	IT_PROG	21-MAY-91	60
107	Lorentz	IT_PROG	07-FEB-99	60
124	Mourgos	ST_MAN	16-NOV-99	50
141	Rajs	ST_CLERK	17-OCT-95	50
142	Davies	ST_CLERK	29-JAN-97	50
143	Matos	ST_CLERK	15-MAR-98	50
144	Vargas	ST_CLERK	09-JUL-98	50
149	Zlotkey	SA_MAN	29-JAN-00	80
174	Abel	SA_REP	11-MAY-96	80
176	Taylor	SA_REP	24-MAR-98	80
EMPLOYEE_ID	LAST_NAME	JOB_ID	HIRE_DATE	DEPARTMENT_ID
178	Grant	SA_REP	24-MAY-99	
200	Whalen	AD_ASST	17-SEP-87	10
201	Hartstein	MK_MAN	17-FEB-96	20

...

Tables Used in This Lesson (continued)

```
DESC job_history
```

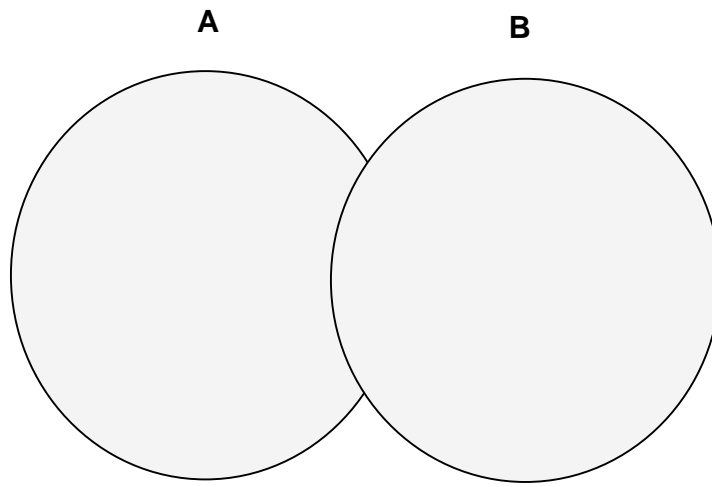
Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
START_DATE	NOT NULL	DATE
END_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
DEPARTMENT_ID		NUMBER(4)

```
SELECT * FROM job_history;
```

EMPLOYEE_ID	START_DAT	END_DATE	JOB_ID	DEPARTMENT_ID
102	13-JAN-93	24-JUL-98	IT_PROG	60
101	21-SEP-89	27-OCT-93	AC_ACCOUNT	110
101	28-OCT-93	15-MAR-97	AC_MGR	110
201	17-FEB-96	19-DEC-99	MK_REP	20
114	24-MAR-98	31-DEC-99	ST_CLERK	50
122	01-JAN-99	31-DEC-99	ST_CLERK	50
200	17-SEP-87	17-JUN-93	AD_ASST	90
176	24-MAR-98	31-DEC-98	SA_REP	80
176	01-JAN-99	31-DEC-99	SA_MAN	80
200	01-JUL-94	31-DEC-98	AC_ACCOUNT	90

10 rows selected.

The UNION Operator



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

ORACLE

1-7

Copyright © Oracle Corporation, 2001. All rights reserved.

The UNION Operator

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

Guidelines

- The number of columns and the datatypes of the columns being selected must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.
- UNION operates over all of the columns being selected.
- NULL values are not ignored during duplicate checking.
- The IN operator has a higher precedence than the UNION operator.
- By default, the output is sorted in ascending order of the first column of the SELECT clause.

Using the UNION Operator

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

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

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

ORACLE

1-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Using the UNION SET Operator

The UNION operator eliminates any duplicate records. If there are records that occur both in the EMPLOYEES and the JOB_HISTORY tables and are identical, the records will be displayed only once. Observe in the output shown on the slide that the record for the employee with the EMPLOYEE_ID 200 appears twice as the JOB_ID is different in each row.

Consider the following example:

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

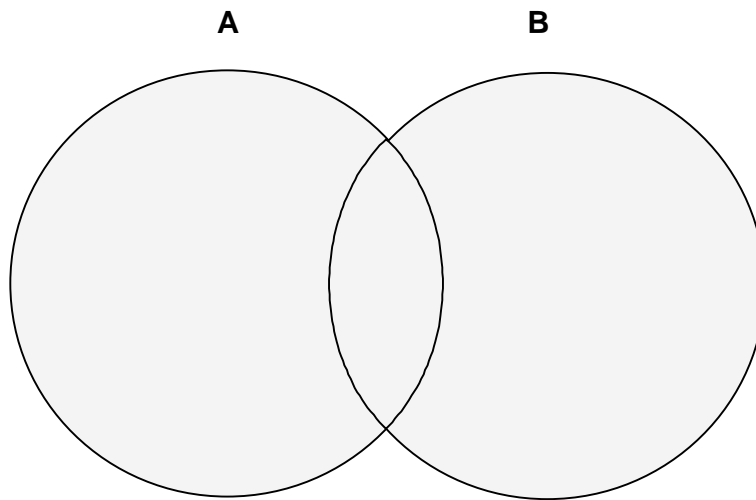
EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
...		
200	AC_ACCOUNT	90
200	AD_ASST	10
200	AD_ASST	90
...		

29 rows selected.

Using the UNION SET Operator (continued)

In the preceding output, employee 200 appears three times. Why? Notice the `DEPARTMENT_ID` values for employee 200. One row has a `DEPARTMENT_ID` of 90, another 10, and the third 90. Because of these unique combinations of job IDs and department IDs, each row for employee 200 is unique and therefore not considered a duplicate. Observe that the output is sorted in ascending order of the first column of the `SELECT` clause, `EMPLOYEE_ID` in this case.

The UNION ALL Operator



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

ORACLE

1-10

Copyright © Oracle Corporation, 2001. All rights reserved.

The UNION ALL Operator

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

Guidelines

- Unlike UNION, duplicate rows are not eliminated and the output is not sorted by default.
- The DISTINCT keyword cannot be used.

Note: With the exception of the above, the guidelines for UNION and UNION ALL are the same.

Using the UNION ALL Operator

Display the current and previous departments of all employees.

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

EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
100	AD_PRES	90
101	AD_VP	90
...		
200	AD_ASST	10
200	AD_ASST	90
200	AC_ACCOUNT	90
...		
205	AC_MGR	110
206	AC_ACCOUNT	110

30 rows selected.

ORACLE

1-11

Copyright © Oracle Corporation, 2001. All rights reserved.

The UNION ALL Operator (continued)

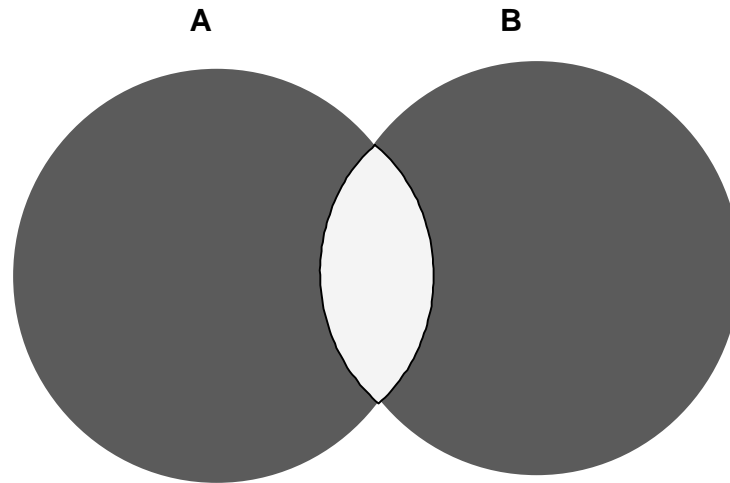
In the example, 30 rows are selected. The combination of the two tables totals to 30 rows. The UNION ALL operator does not eliminate duplicate rows. The duplicate rows are highlighted in the output shown in the slide. UNION returns all distinct rows selected by either query. UNION ALL returns all rows selected by either query, including all duplicates. Consider the query on the slide, now written with the UNION clause:

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

The preceding query returns 29 rows. This is because it eliminates the following row (as it is a duplicate):

EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
176	SA_REP	80

The INTERSECT Operator



ORACLE

1-12

Copyright © Oracle Corporation, 2001. All rights reserved.

The INTERSECT Operator

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

Guidelines

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

Using the INTERSECT Operator

Display the employee IDs and job IDs of employees who currently have a job title that they held before beginning 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

ORACLE

1-13

Copyright © Oracle Corporation, 2001. All rights reserved.

The INTERSECT Operator (continued)

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

What will be the results if you add the DEPARTMENT_ID column to the SELECT statement from the EMPLOYEES table and add the DEPARTMENT_ID column to the SELECT statement from the JOB_HISTORY table and run this query? The results may be different because of the introduction of another column whose values may or may not be duplicates.

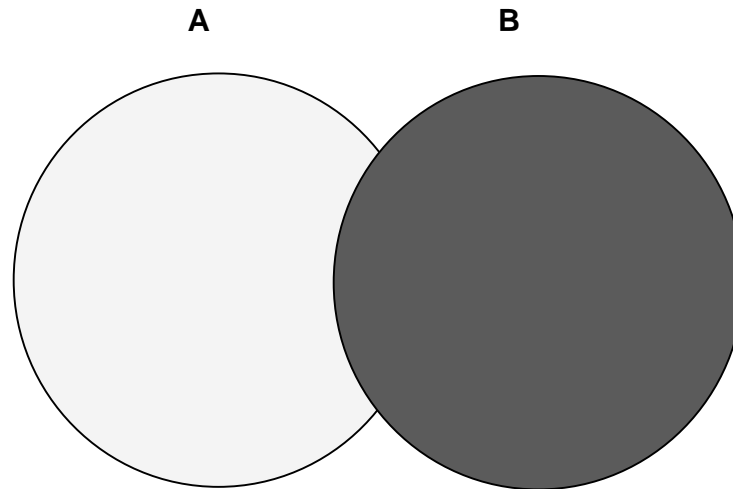
Example

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

EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
176	SA_REP	80

Employee 200 is no longer part of the results because the EMPLOYEES.DEPARTMENT_ID value is different from the JOB_HISTORY.DEPARTMENT_ID value.

The MINUS Operator



ORACLE

1-14

Copyright © Oracle Corporation, 2001. All rights reserved.

The MINUS Operator

Use the MINUS operator to return rows returned by the first query that are not present in the second query (the first SELECT statement MINUS the second SELECT statement).

Guidelines

- The number of columns and the datatypes of the columns being selected by the SELECT statements in the queries must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.
- All of the columns in the WHERE clause must be in the SELECT clause for the MINUS operator to work.

The MINUS Operator

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

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

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

18 rows selected.

ORACLE

The MINUS Operator (continued)

In the example in the slide, the employee IDs and Job IDs in the JOB_HISTORY table are subtracted from those in the EMPLOYEES table. The results set displays the employees remaining after the subtraction; they are represented by rows that exist in the EMPLOYEES table but do not exist in the JOB_HISTORY table. These are the records of the employees who have not changed their jobs even once.

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

ORACLE

1-16

Copyright © Oracle Corporation, 2001. All rights reserved.

SET Operator Guidelines

- The expressions in the select lists of the queries must match in number and datatype. Queries that use **UNION**, **UNION ALL**, **INTERSECT**, and **MINUS** SET operators in their **WHERE** clause must have the same number and type of columns in their **SELECT** list. For example:

```
SELECT employee_id, department_id
FROM   employees
WHERE  (employee_id, department_id)
       IN (SELECT employee_id, department_id
           FROM   employees
           UNION
           SELECT employee_id, department_id
           FROM   job_history);
```
- The **ORDER BY** clause:
 - Can appear only at the very end of the statement
 - Will accept the column name, an alias, or the positional notation
- The column name or alias, if used in an **ORDER BY** clause, must be from the first **SELECT** list.
- SET operators can be used in subqueries.

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.**

ORACLE

1-17

Copyright © Oracle Corporation, 2001. All rights reserved.

The Oracle Server and SET Operators

When a query uses SET operators, the Oracle Server eliminates duplicate rows automatically except in the case of the UNION ALL operator. The column names in the output are decided by the column list in the first SELECT statement. By default, the output is sorted in ascending order of the first column of the SELECT clause.

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

- If both queries select values of datatype CHAR, the returned values have datatype CHAR.
- If either or both of the queries select values of datatype VARCHAR2, the returned values have datatype VARCHAR2.

Matching the SELECT Statements

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

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

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

27 rows selected.

ORACLE

Matching the SELECT Statements

As the expressions in the select lists of the queries must match in number, you can use dummy columns and the datatype conversion functions to comply with this rule. In the slide, the name `location` is given as the dummy column heading. The `TO_NUMBER` function is used in the first query to match the `NUMBER` datatype of the `LOCATION_ID` column retrieved by the second query. Similarly, the `TO_DATE` function in the second query is used to match the `DATE` datatype of the `HIRE_DATE` column retrieved by the first query.

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
...		
205	AC_MGR	12000
206	AC_ACCOUNT	8300

30 rows selected.

ORACLE

Matching the SELECT Statement: Example

The EMPLOYEES and JOB_HISTORY tables have several columns in common; for example, EMPLOYEE_ID, JOB_ID and DEPARTMENT_ID. But what if you want the query to display the EMPLOYEE_ID, JOB_ID, and SALARY using the UNION operator, knowing that the salary exists only in the, EMPLOYEES table?

The code example in the slide matches the EMPLOYEE_ID and the JOB_ID columns in the EMPLOYEES and in the JOB_HISTORY tables. A literal value of 0 is added to the JOB_HISTORY SELECT statement to match the numeric SALARY column in the EMPLOYEES SELECT statement.

In the preceding results, each row in the output that corresponds to a record from the JOB_HISTORY table contains a 0 in the SALARY column.

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

ORACLE

1-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Controlling the Order of Rows

By default, the output is sorted in ascending order on the first column. You can use the ORDER BY clause to change this.

Using ORDER BY to Order Rows

The ORDER BY clause can be used only once in a compound query. If used, the ORDER BY clause must be placed at the end of the query. The ORDER BY clause accepts the column name, an alias, or the positional notation. Without the ORDER BY clause, the code example in the slide produces the following output in the alphabetical order of the first column:

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

Note: Consider a compound query where the UNION SET operator is used more than once. In this case, the ORDER BY clause can use only positions rather than explicit expressions.

Summary

In this lesson, you should have learned how to:

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

ORACLE

1-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

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

Practice 1 Overview

This practice covers using the Oracle9i datetime functions.

ORACLE

1-22

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 1 Overview

In this practice, you write queries using the SET operators.

Practice 1

1. List the department IDs for departments that do not contain the job ID ST_CLERK, using SET operators.

DEPARTMENT_ID
10
20
60
80
90
110
190

7 rows selected.

2. Display the country ID and the name of the countries that have no departments located in them, using SET operators.

CO	COUNTRY_NAME
DE	Germany

3. Produce a list of jobs for departments 10, 50, and 20, in that order. Display job ID and department ID, using SET operators.

JOB_ID	DEPARTMENT_ID
AD_ASST	10
ST_CLERK	50
ST_MAN	50
MK_MAN	20
MK_REP	20

4. List the employee IDs and job IDs of those employees who currently hold the job title that they held before beginning their tenure with the company.

EMPLOYEE_ID	JOB_ID
176	SA_REP
200	AD_ASST

Practice 1 (continued)

5. Write a compound query that lists the following:
 - Last names and department ID of all the employees from the EMPLOYEES table, regardless of whether or not they belong to any department or not
 - Department ID and department name of all the departments from the DEPARTMENTS table, regardless of whether or not they have employees working in them

LAST_NAME	DEPARTMENT_ID	TO_CHAR(NULL)
Abel	80	
Davies	50	
De Haan	90	
Ernst	60	
Fay	20	
Gietz	110	
Grant		
Hartstein	20	
Higgins	110	
Hunold	60	
King	90	
Kochhar	90	
Lorentz	60	
Matos	50	
LAST_NAME	DEPARTMENT_ID	TO_CHAR(NULL)
Mourgos	50	
Rajs	50	
Taylor	80	
Vargas	50	
Whalen	10	
Zlotkey	80	
	10	Administration
	20	Marketing
	50	Shipping
	60	IT
	80	Sales
	90	Executive
	110	Accounting
	190	Contracting

28 rows selected.

2

Oracle9i Datetime Functions

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

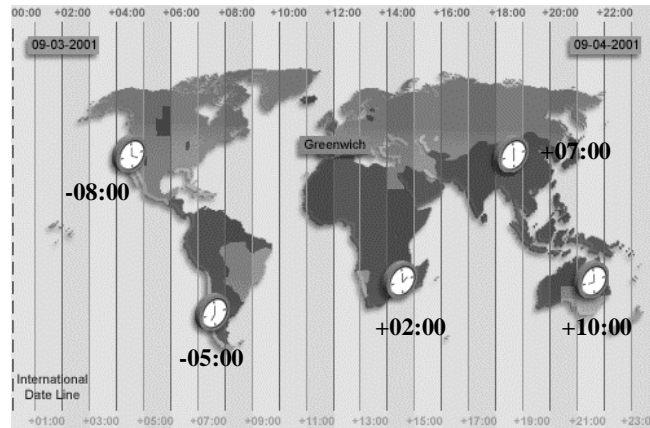
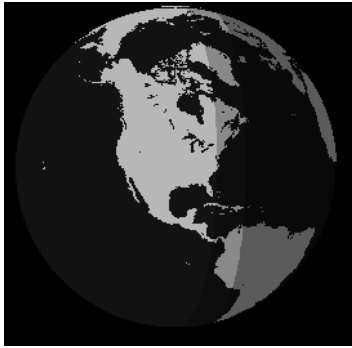
- TZ_OFFSET
- FROM_TZ
- TO_TIMESTAMP
- TO_TIMESTAMP_TZ
- TO_YMINTERVAL
- CURRENT_DATE
- CURRENT_TIMESTAMP
- LOCALTIMESTAMP
- DBTIMEZONE
- SESSIONTIMEZONE
- EXTRACT

ORACLE

Lesson Aim

This lesson addresses some of the datetime functions introduced in Oracle9i.

TIME ZONES



The image represents the time for each time zone when Greenwich time is 12:00.

ORACLE

2-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Time Zones

In Oracle9i, you can include the time zone in your date and time data, as well as provide support for fractional seconds. This lesson focuses on how to manipulate the new datetime data types included with Oracle9i using the new datetime functions. To understand the working of these functions, it is necessary to be familiar with the concept of time zones and Greenwich Mean Time, or GMT. Greenwich Mean Time, or GMT is now referred to as UTC (Coordinated Universal Time).

The hours of the day are measured by the turning of the earth. The time of day at any particular moment depends on where you are. When it is noon in Greenwich, England, it is midnight along the international date line. The earth is divided into 24 time zones, one for each hour of the day. The time along the prime meridian in Greenwich, England is known as Greenwich mean time, or GMT. GMT is the time standard against which all other time zones in the world are referenced. It is the same all year round and is not effected by summer time or daylight savings time. The meridian line is an imaginary line that runs from the North Pole to the South Pole. It is known as zero longitude and it is the line from which all other lines of longitude are measured. All time is measured relative to Greenwich mean time (GMT) and all places have a latitude (their distance north or south of the equator) and a longitude (their distance east or west of the Greenwich meridian).

Daylight Saving Time

Most western nations advance the clock ahead one hour during the summer months. This period is called daylight saving time. Daylight saving time lasts from the first Sunday in April to the last Sunday in October in the most of the United States, Mexico and Canada. The nations of the European Union observe daylight saving time, but they call it the summer time period. Europe's summer time period begins a week earlier than its North American counterpart, but ends at the same time.

Oracle9i Datetime Support

- In Oracle9i, 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)`
- Oracle9i provides daylight savings support for datetime data types in the server.

ORACLE

2-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle9i Datetime Support

With Oracle9i, three new data types are added to DATE, with the following differences:

Data Type	Time Zone	Fractional Seconds
DATE	No	No
TIMESTAMP	No	Yes
TIMESTAMP (<i>fractional_seconds_precision</i>) WITH TIMEZONE	All values of TIMESTAMP as well as the time zone displacement value which indicates the hours and minutes before or after UTC (Coordinated Universal Time, formerly Greenwich mean time).	<i>fractional_seconds_precision</i> is the number of digits in the fractional part of the SECOND datetime field. Accepted values are 0 to 9. The default is 6.
TIMESTAMP (<i>fractional_seconds_precision</i>) WITH LOCAL TIME ZONE	All values of TIMESTAMP WITH TIME ZONE, with the following exceptions: <ul style="list-style-type: none">• Data is normalized to the database time zone when it is stored in the database.• When the data is retrieved, users see the data in the session time zone.	Yes

Oracle9i Datetime Support (continued)

`TIMESTAMP WITH LOCAL TIME ZONE` is stored in the database time zone. When a user selects the data, the value is adjusted to the user's session time zone.

Example:

A San Francisco database has system time zone = -8:00. When a New York client (session time zone = -5:00) inserts into or selects from the San Francisco database, `TIMESTAMP WITH LOCAL TIME ZONE` data is adjusted as follows:

- The New York client inserts `TIMESTAMP '1998-1-23 6:00:00-5:00'` into a `TIMESTAMP WITH LOCAL TIME ZONE` column in the San Francisco database. The inserted data is stored in San Francisco as binary value `1998-1-23 3:00:00`.
- When the New York client selects that inserted data from the San Francisco database, the value displayed in New York is `'1998-1-23 6:00:00'`.
- A San Francisco client, selecting the same data, see the value `'1998-1-23 3:00:00'`.

Support for Daylight Savings Times

The Oracle Server automatically determines, for any given time zone region, whether daylight savings is in effect and returns local time values based accordingly. The datetime value is sufficient for the server to determine whether daylight savings time is in effect for a given region in all cases except boundary cases. A boundary case occurs during the period when daylight savings goes into or comes out of effect. For example, in the U.S.-Pacific region, when daylight savings comes into effect, the time changes from 2:00 a.m. to 3:00 a.m. The one hour interval between 2:00 a.m. and 3:00 a.m. does not exist. When daylight savings goes out of effect, the time changes from 2:00 a.m. back to 1:00 a.m., and the one-hour interval between 1:00 a.m. and 2:00 a.m. is repeated.

Oracle9i also significantly reduces the cost of developing and deploying applications globally on a single database instance. Requirements for multigeographic applications include named time zones and multilanguage support through Unicode. The datetime data types `TSLTZ` and `TSTZ` are time-zone-aware. Datetime values can be specified as local time in a particular region (rather than a particular offset). Using the time zone rules tables for a given region, the time zone offset for a local time is calculated, taking into consideration daylight savings time adjustments, and used in further operations.

This lesson addresses some of the new datetime functions introduced in Oracle9i.

TZ_OFFSET

- Display the time zone offset for the time zone 'US/Eastern'

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

TZ_OFFSET
-04:00

- Display the time zone offset for the time zone 'Canada/Yukon'

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

TZ_OFFSET
-07:00

- Display the time zone offset for the time zone 'Europe/London'

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

TZ_OFFSET
+01:00

ORACLE

2-6

Copyright © Oracle Corporation, 2001. All rights reserved.

TZ_OFFSET

The TZ_OFFSET function returns the time zone offset corresponding to the value entered. The return value is dependent on the date when the statement is executed. For example if the TZ_OFFSET function returns a value -08:00, the return value can be interpreted as the time zone from where the command was executed is eight hours after UTC. You can enter a valid time zone name, a time zone offset from UTC (which simply returns itself), or the keyword SESSIONTIMEZONE or DBTIMEZONE. The syntax of the TZ_OFFSET function is:

```
TZ_OFFSET ( ['time_zone_name'] '[+ | -] hh:mm'
            [ SESSIONTIMEZONE] [DBTIMEZONE])
```

The examples in the slide can be interpreted as follows:

- The time zone 'US/Eastern' is four hours behind UTC
- The time zone 'Canada/Yukon' is seven hours behind UTC
- The time zone 'Europe/London' is one hour ahead of UTC

For a listing of valid time zone name values, query the V\$TIMEZONE_NAMES dynamic performance view.

```
DESC V$TIMEZONE_NAMES
```

Name	Null?	Type
TZNAME		VARCHAR2(64)
TZABBREV		VARCHAR2(64)

TZ_OFFSET (continued)

```
SELECT * FROM V$TIMEZONE_NAMES;
```

TZNAME	TZABBREV
Africa/Cairo	LMT
Africa/Cairo	EET
Africa/Cairo	EEST
Africa/Tripoli	LMT
Africa/Tripoli	CET
Africa/Tripoli	CEST
Africa/Tripoli	EET
America/Adak	LMT
America/Adak	NST
America/Adak	NWT
America/Adak	BST
America/Adak	BDT
America/Adak	HAST
America/Adak	HADT
TZNAME	TZABBREV
America/Anchorage	LMT
America/Anchorage	CAT
America/Anchorage	CAWT
America/Anchorage	AHST
America/Anchorage	AHDT
America/Anchorage	AKST
■ ■ ■	
W-SU	MDST
W-SU	S
W-SU	MSD
W-SU	MSK
W-SU	EET
W-SU	EEST
WET	WEST
WET	WET

616 rows selected.

CURRENT_DATE

- Display the current date and time in the session's time zone .

```
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	03-OCT-2001 09:37:06

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

SESSIONTIMEZONE	CURRENT_DATE
-08:00	03-OCT-2001 06:38:07

- **CURRENT_DATE** is sensitive to the session time zone.
- The return value is a date in the Gregorian calendar.

ORACLE

2-8

Copyright © Oracle Corporation, 2001. All rights reserved.

CURRENT_DATE

The **CURRENT_DATE** function returns the current date in the session's time zone. The return value is a date in the Gregorian calendar.

The examples in the slide illustrate that **CURRENT_DATE** is sensitive to the session time zone. In the first example, the session is altered to set the **TIME_ZONE** parameter to **-5:0**. The **TIME_ZONE** parameter specifies the default local time zone displacement for the current SQL session. **TIME_ZONE** is a session parameter only, not an initialization parameter. The **TIME_ZONE** parameter is set as follows:

```
TIME_ZONE = '[+ | -] hh:mm'
```

The format mask (**[+ | -] hh:mm**) indicates the hours and minutes before or after UTC (Coordinated Universal Time, formerly known as Greenwich mean time).

Observe in the output that the value of **CURRENT_DATE** changes when the **TIME_ZONE** parameter value is changed to **-8:0** in the second example.

Note: The **ALTER SESSION** command sets the date format of the session to **'DD-MON-YYYY HH24:MI:SS'** that is Day of month (1-31)-Abbreviated name of month-4-digit year Hour of day (0-23):Minute (0-59):Second (0-59).

CURRENT_TIMESTAMP

- Display the current date and fractional time in the session's time zone.

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

SESSIONTIMEZONE	CURRENT_TIMESTAMP
-05:00	03-OCT-01 09.40.59.000000 AM -05:00

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

SESSIONTIMEZONE	CURRENT_TIMESTAMP
-08:00	03-OCT-01 06.41.38.000000 AM -08:00

- **CURRENT_TIMESTAMP is sensitive to the session time zone.**
- **The return value is of the TIMESTAMP WITH TIME ZONE datatype.**

ORACLE

CURRENT_TIMESTAMP

The `CURRENT_TIMESTAMP` function returns the current date and time in the session time zone, as a value of the data type `TIMESTAMP WITH TIME ZONE`. The time zone displacement reflects the current local time of the SQL session. The syntax of the `CURRENT_TIMESTAMP` function is:

`CURRENT_TIMESTAMP (precision)`

where *precision* is an optional argument that specifies the fractional second precision of the time value returned. If you omit precision, the default is 6.

The examples in the slide illustrates that `CURRENT_TIMESTAMP` is sensitive to the session time zone. In the first example, the session is altered to set the `TIME_ZONE` parameter to `-5:0`. Observe in the output that the value of `CURRENT_TIMESTAMP` changes when the `TIME_ZONE` parameter value is changed to `-8:0` in the second example.

LOCALTIMESTAMP

- Display the current date and time in the session's time zone in a value of **TIMESTAMP** data type.

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

CURRENT_TIMESTAMP	LOCALTIMESTAMP
03-OCT-01 09.44.21.000000 AM -05:00	03-OCT-01 09.44.21.000000 AM

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

CURRENT_TIMESTAMP	LOCALTIMESTAMP
03-OCT-01 06.45.21.000001 AM -08:00	03-OCT-01 06.45.21.000001 AM

- **LOCALTIMESTAMP** returns a **TIMESTAMP** value, whereas **CURRENT_TIMESTAMP** returns a **TIMESTAMP WITH TIME ZONE** value.

ORACLE

LOCALTIMESTAMP

The **LOCALTIMESTAMP** function returns the current date and time in the session time zone in a value of data type **TIMESTAMP**. The difference between this function and **CURRENT_TIMESTAMP** is that **LOCALTIMESTAMP** returns a **TIMESTAMP** value, while **CURRENT_TIMESTAMP** returns a **TIMESTAMP WITH TIME ZONE** value. **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. The **TIMESTAMP WITH TIME ZONE** data type has the following format:

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

where *fractional_seconds_precision* optionally specifies the number of digits in the fractional part of the **SECOND** datetime field and can be a number in the range 0 to 9. The default is 6. For example, you specify **TIMESTAMP WITH TIME ZONE** as a literal as follows:

TIMESTAMP '1997-01-31 09:26:56.66 +02:00'

The syntax of the **LOCAL_TIMESTAMP** function is:

LOCAL_TIMESTAMP (*TIMESTAMP_precision*)

Where, *TIMESTAMP_precision* is an optional argument that specifies the fractional second precision of the **TIMESTAMP** value returned.

The examples in the slide illustrates the difference between **LOCALTIMESTAMP** and **CURRENT_TIMESTAMP**. Observe that the **LOCALTIMESTAMP** does not display the time zone value, while the **CURRENT_TIMESTAMP** does.

DBTIMEZONE and SESSIONTIMEZONE

- Display the value of the database time zone.

```
SELECT DBTIMEZONE FROM DUAL;
```

DBTIME
-05:00

- Display the value of the session's time zone.

```
SELECT SESSIONTIMEZONE FROM DUAL;
```

SESSIONTIMEZONE
-08:00

ORACLE

2-11

Copyright © Oracle Corporation, 2001. All rights reserved.

DBTIMEZONE and SESSIONTIMEZONE

The default database time zone is the same as the operating system's time zone. You set the database's default time zone by specifying the `SET TIME_ZONE` clause of the `CREATE DATABASE` statement. If omitted, the default database time zone is the operating system time zone. The database time zone can be changed for a session with an `ALTER SESSION` statement.

The `DBTIMEZONE` function returns the value of the database time zone. The return type is a time zone offset (a character type in the format '`[+ | -]TZH:TZM`') or a time zone region name, depending on how the user specified the database time zone value in the most recent `CREATE DATABASE` or `ALTER DATABASE` statement. The example on the slide shows that the database time zone is set to UTC, as the `TIME_ZONE` parameter is in the format:

```
TIME_ZONE = '[+ | -] hh:mm'
```

The `SESSIONTIMEZONE` function returns the value of the current session's time zone. The return type is a time zone offset (a character type in the format '`[+ | -]TZH:TZM`') or a time zone region name, depending on how the user specified the session time zone value in the most recent `ALTER SESSION` statement. The example in the slide shows that the session time zone is set to UTC.

Observe that the database time zone is different from the current session's time zone.

EXTRACT

- Display the YEAR component from the SYSDATE.

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

EXTRACT(YEARFROMSYSDATE)
2001

- Display the MONTH component from the HIRE_DATE for those employees whose MANAGER_ID is 100.

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

LAST_NAME	HIRE_DATE	EXTRACT(MONTHFROMHIRE_DATE)
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

ORACLE

2-12

Copyright © Oracle Corporation, 2001. All rights reserved.

EXTRACT

The EXTRACT expression extracts and returns the value of a specified datetime field from a datetime or interval value expression. You can extract any of the components mentioned in the following syntax using the EXTRACT function. The syntax of the EXTRACT function is:

```
SELECT  EXTRACT ([YEAR] [MONTH][DAY] [HOUR] [MINUTE][SECOND]  
               [TIMEZONE_HOUR] [TIMEZONE_MINUTE]  
               [TIMEZONE_REGION] [TIMEZONE_ABBR]  
FROM    [datetime_value_expression]  
        [interval_value_expression]);
```

When you extract a TIMEZONE_REGION or TIMEZONE_ABBR (abbreviation), the value returned is a string containing the appropriate time zone name or abbreviation. When you extract any of the other values, the value returned is in the Gregorian calendar. When extracting from a datetime with a time zone value, the value returned is in UTC. For a listing of time zone names and their corresponding abbreviations, query the V\$TIMEZONE_NAMES dynamic performance view. In the first example on the slide, the EXTRACT function is used to extract the YEAR from SYSDATE .

In the second example in the slide, the EXTRACT function is used to extract the MONTH from HIRE_DATE column of the EMPLOYEES table, for those employees who report to the manager whose EMPLOYEE_ID is 100.

TIMESTAMP Conversion Using FROM_TZ

- Display the **TIMESTAMP** value '2000-03-28 08:00:00' as a **TIMESTAMP WITH TIME ZONE** value.

```
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
```

- Display the **TIMESTAMP** value '2000-03-28 08:00:00' as a **TIMESTAMP WITH TIME ZONE** value for the time zone region 'Australia/North'

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

```
FROM_TZ(TIMESTAMP'2000-03-2808:00:00','AUSTRALIA/NORTH')
28-MAR-00 08.00.00.000000000 AM AUSTRALIA/NORTH
```

ORACLE

TIMESTAMP Conversion Using FROM_TZ

The `FROM_TZ` function converts a `TIMESTAMP` value to a `TIMESTAMP WITH TIME ZONE` value.

The syntax of the `FROM_TZ` function is as follows:

```
FROM_TZ(TIMESTAMP timestamp_value, time_zone_value)
```

where *time_zone_value* is a character string in the format 'TZH:TZM' or a character expression that returns a string in TZR (time zone region) with optional TZD format (TZD is an abbreviated time zone string with daylight savings information.) TZR represents the time zone region in datetime input strings. Examples are 'Australia/North', 'UTC', and 'Singapore'. TZD represents an abbreviated form of the time zone region with daylight savings information. Examples are 'PST' for US/Pacific standard time and 'PDT' for US/Pacific daylight time. To see a listing of valid values for the TZR and TZD format elements, query the `V$TIMEZONE_NAMES` dynamic performance view.

The example in the slide converts a `TIMESTAMP` value to `TIMESTAMP WITH TIME ZONE`.

STRING To TIMESTAMP Conversion Using TO_TIMESTAMP and TO_TIMESTAMP_TZ

- Display the character string '2000-12-01 11:00:00' as a **TIMESTAMP** value.

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

```
TO_TIMESTAMP('2000-12-01 11:00:00','YYYY-MM-DDHH:MI:SS')  
01-DEC-00 11.00.00.000000000 AM
```

- Display the character string '1999-12-01 11:00:00 -8:00' as a **TIMESTAMP WITH TIME ZONE** value.

```
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('1999-12-01 11:00:00-8:00','YYYY-MM-DDHH:MI:SSTZH:TZM')  
01-DEC-99 11.00.00.000000000 AM -08:00
```

ORACLE

STRING To TIMESTAMP Conversion Using TO_TIMESTAMP and TO_TIMESTAMP_TZ

The `TO_TIMESTAMP` function converts a string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type to a value of `TIMESTAMP` data type. The syntax of the `TO_TIMESTAMP` function is:

```
TO_TIMESTAMP ( char, [fmt], [ 'nlsparam' ] )
```

The optional *fmt* specifies the format of *char*. If you omit *fmt*, the string must be in the default format of the `TIMESTAMP` data type. The optional *nlsparam* specifies the language in which month and day names and abbreviations are returned. This argument can have this form:

```
'NLS_DATE_LANGUAGE = language'
```

If you omit *nlsparams*, this function uses the default date language for your session. The example on the slide converts a character string to a value of `TIMESTAMP`.

The `TO_TIMESTAMP_TZ` function converts a string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type to a value of `TIMESTAMP WITH TIME ZONE` data type. The syntax of the `TO_TIMESTAMP_TZ` function is:

```
TO_TIMESTAMP_TZ ( char, [fmt], [ 'nlsparam' ] )
```

The optional *fmt* specifies the format of *char*. If omitted, a string must be in the default format of the `TIMESTAMP WITH TIME ZONE` data type. The optional *nlsparam* has the same purpose in this function as in the `TO_TIMESTAMP` function. The example in the slide converts a character string to a value of `TIMESTAMP WITH TIME ZONE`.

Note: The `TO_TIMESTAMP_TZ` function does not convert character strings to `TIMESTAMP WITH LOCAL TIME ZONE`.

Time Interval Conversion with TO_YMINTERVAL

- Display a date that is one year two months after the hire date for the employees working in the department with the DEPARTMENT_ID 20

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

HIRE_DATE	HIRE_DATE_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

ORACLE

2-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Time Interval Conversion with TO_YMINTERVAL

The TO_YMINTERVAL function converts a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to an INTERVAL YEAR TO MONTH data type. The INTERVAL YEAR TO MONTH data type stores a period of time using the YEAR and MONTH datetime fields. The format of INTERVAL YEAR TO MONTH is as follows:

INTERVAL YEAR [(*year_precision*)] TO MONTH

where *year_precision* is the number of digits in the YEAR datetime field. The default value of *year_precision* is 2.

The syntax of the TO_YMINTERVAL function is:

TO_YMINTERVAL (*char*)

where *char* is the character string to be converted.

The example in the slide calculates a date that is one year two months after the hire date for the employees working in the department 20 of the EMPLOYEES table.

A reverse calculation can also be done using the TO_YMINTERVAL function. For example:

```
SELECT hire_date, hire_date + TO_YMINTERVAL('-02-04') AS  
       HIRE_DATE_YMININTERVAL  
FROM   EMPLOYEES WHERE department_id = 20;
```

Observe that the character string passed to the TO_YMINTERVAL function has a negative value. The example returns a date that is two years and four months before the hire date for the employees working in the department 20 of the EMPLOYEES table.

Summary

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

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

ORACLE

Summary

This lesson addressed some of the new datetime functions introduced in Oracle9i.

Practice 2 Overview

This practice covers using the Oracle9i datetime functions.

ORACLE

2-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 2 Overview

In this practice, you display time zone offsets, `CURRENT_DATE`, `CURRENT_TIMESTAMP`, and the `LOCALTIMESTAMP`. You also set time zones and use the `EXTRACT` function.

Practice 2

1. Alter the session to set the NLS_DATE_FORMAT to DD-MON-YYYY HH24:MI:SS.
2. a. Write queries to display the time zone offsets (TZ_OFFSET), for the following time zones.

– *US/Pacific-New*

TZ_OFFSET
-07:00

– *Singapore*

TZ_OFFSET
+08:00

– *Egypt*

TZ_OFFSET
+02:00

- b. Alter the session to set the TIME_ZONE parameter value to the time zone offset of US/Pacific-New.
- c. Display the CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP for this session.

Note: The output might be different based on the date when the command is executed.

CURRENT_DATE	CURRENT_TIMESTAMP	LOCALTIMESTAMP
01-OCT-2001 13:40:54	01-OCT-01 01.40.54.000001 PM -07:00	01-OCT-01 01.40.54.000001 PM

- d. Alter the session to set the TIME_ZONE parameter value to the time zone offset of Singapore.
- e. Display the CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP for this session. **Note:** The output might be different based on the date when the command is executed.

CURRENT_DATE	CURRENT_TIMESTAMP	LOCALTIMESTAMP
02-OCT-2001 04:42:34	02-OCT-01 04.42.34.000000 AM +08:00	02-OCT-01 04.42.34.000000 AM

Note: Observe in the preceding practice that CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP are all sensitive to the session time zone.

3. Write a query to display the DBTIMEZONE and SESSIONTIMEZONE.

DBTIMEZONE	SESSIONTIMEZONE
-05:00	+08:00

Practice 2 (continued)

4. Write a query to extract the YEAR from HIRE_DATE column of the EMPLOYEES table for those employees who work in department 80.

LAST_NAME	EXTRACT(YEARFROMHIRE_DATE)
Zlotkey	2000
Abel	1996
Taylor	1998

5. Alter the session to set the NLS_DATE_FORMAT to DD-MON-YYYY.

3

Enhancements to the GROUP BY Clause

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

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**

ORACLE

3-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this lesson you learn how to:

- Group data for obtaining the following:
 - Subtotal values by using the ROLLUP operator
 - Cross-tabulation values by using the CUBE operator
- Use the GROUPING function to identify the level of aggregation in the results set produced by a ROLLUP or CUBE operator.
- Use GROUPING SETS to produce a single result set that is equivalent to a UNION ALL approach.

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%';
```

ORACLE

3-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Group Functions

You can use the `GROUP BY` clause to divide the rows in a table into groups. You can then use the group functions to return summary information for each group. Group functions can appear in select lists and in `ORDER BY` and `HAVING` clauses. The Oracle Server applies the group functions to each group of rows and returns a single result row for each group.

Types of Group Functions

Each of the group functions `AVG`, `SUM`, `MAX`, `MIN`, `COUNT`, `STDDEV`, and `VARIANCE` accept one argument. The functions `AVG`, `SUM`, `STDDEV`, and `VARIANCE` operate only on numeric values. `MAX` and `MIN` can operate on numeric, character, or date data values. `COUNT` returns the number of nonnull rows for the given expression. The example in the slide calculates the average salary, standard deviation on the salary, number of employees earning a commission and the maximum hire date for those employees whose `JOB_ID` begins with `SA`.

Guidelines for Using Group Functions

- The data types for the arguments can be `CHAR`, `VARCHAR2`, `NUMBER`, or `DATE`.
- All group functions except `COUNT (*)` ignore null values. To substitute a value for null values, use the `NVL` function. `COUNT` returns either a number or zero.
- The Oracle Server implicitly sorts the results set in ascending order of the grouping columns specified, when you use a `GROUP BY` clause. To override this default ordering, you can use `DESC` in an `ORDER BY` clause.

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 ;
```

ORACLE

3-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Review of GROUP BY Clause

The example illustrated in the slide is evaluated by the Oracle Server as follows:

- The SELECT clause specifies that the following columns are to be retrieved:
 - Department ID and job ID columns from the EMPLOYEES table
 - The sum of all the salaries and the number of employees in each group that you have specified in the GROUP BY clause
- The GROUP BY clause specifies how the rows should be grouped in the table. The total salary and the number of employees are calculated for each job ID within each department. The rows are grouped by department ID and then grouped by job within each department.

DEPARTMENT_ID	JOB_ID	SUM(SALARY)	COUNT(EMPLOYEE_ID)
10	AD_ASST	4400	1
20	MK_MAN	13000	1
20	MK_REP	6000	1
50	ST_CLERK	11700	4
...			
110	AC_ACCOUNT	8300	1
110	AC_MGR	12000	1
	SA_REP	7000	1

13 rows selected.

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.

ORACLE

3-5

Copyright © Oracle Corporation, 2001. All rights reserved.

The HAVING Clause

Groups are formed and group functions are calculated before the HAVING clause is applied to the groups. The HAVING clause can precede the GROUP BY clause, but it is recommended that you place the GROUP BY clause first because it is more logical.

The Oracle Server performs the following steps when you use the HAVING clause:

1. Groups rows
2. Applies the group functions to the groups and displays the groups that match the criteria in the HAVING clause

```
SELECT department_id, AVG(salary)  
FROM   employees  
GROUP BY department_id  
HAVING AVG(salary) >9500;
```

DEPARTMENT_ID	AVG(SALARY)
80	10033.3333
90	19333.3333
110	10150

The example displays department ID and average salary for those departments whose average salary is greater than \$9,500.

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.**

ORACLE

3-6

Copyright © Oracle Corporation, 2001. All rights reserved.

GROUP BY with the ROLLUP and CUBE Operators

You specify ROLLUP and CUBE operators in the GROUP BY clause of a query. ROLLUP grouping produces a results set containing the regular grouped rows and subtotal rows. The CUBE operation in the GROUP BY clause groups the selected rows based on the values of all possible combinations of expressions in the specification and returns a single row of summary information for each group. You can use the CUBE operator to produce cross-tabulation rows.

Note: When working with ROLLUP and CUBE, make sure that the columns following the GROUP BY clause have meaningful, real-life relationships with each other; otherwise the operators return irrelevant information.

The ROLLUP and CUBE operators are available only in Oracle8i and later releases.

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.

ORACLE

The ROLLUP Operator

The ROLLUP operator delivers aggregates and superaggregates for expressions within a GROUP BY statement. The ROLLUP operator can be used by report writers to extract statistics and summary information from results sets. The cumulative aggregates can be used in reports, charts, and graphs.

The ROLLUP operator creates groupings by moving in one direction, from right to left, along the list of columns specified in the GROUP BY clause. It then applies the aggregate function to these groupings.

Note: To produce subtotals in n dimensions (that is, n columns in the GROUP BY clause) without a ROLLUP operator, $n+1$ SELECT statements must be linked with UNION ALL. This makes the query execution inefficient, because each of the SELECT statements causes table access. The ROLLUP operator gathers its results with just one table access. The ROLLUP operator is useful if there are many columns involved in producing the 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.

ORACLE

3-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of a ROLLUP Operator

In the example in the slide:

- Total salaries for every job ID within a department for those departments whose department ID is less than 60 are displayed by the GROUP BY clause (labeled 1)
- The ROLLUP operator displays:
 - Total salary for those departments whose department ID is less than 60 (labeled 2)
 - Total salary for all departments whose department ID is less than 60, irrespective of the job IDs (labeled 3)
- All rows indicated as 1 are regular rows and all rows indicated as 2 and 3 are superaggregate rows.

The ROLLUP operator creates subtotals that roll up from the most detailed level to a grand total, following the grouping list specified in the GROUP BY clause. First it calculates the standard aggregate values for the groups specified in the GROUP BY clause (in the example, the sum of salaries grouped on each job within a department). Then it creates progressively higher-level subtotals, moving from right to left through the list of grouping columns. (In the preceding example, the sum of salaries for each department is calculated, followed by the sum of salaries for all departments.)

- Given n expressions in the ROLLUP operator of the GROUP BY clause, the operation results in $n + 1 = 2 + 1 = 3$ groupings.
- Rows based on the values of the first n expressions are called rows or regular rows and the others are called superaggregate rows.

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.**

ORACLE

The CUBE Operator

The CUBE operator is an additional switch in the GROUP BY clause in a SELECT statement. The CUBE operator can be applied to all aggregate functions, including AVG, SUM, MAX, MIN, and COUNT. It is used to produce results sets that are typically used for cross-tabular reports. While ROLLUP produces only a fraction of possible subtotal combinations, CUBE produces subtotals for all possible combinations of groupings specified in the GROUP BY clause, and a grand total.

The CUBE operator is used with an aggregate function to generate additional rows in a results set. Columns included in the GROUP BY clause are cross-referenced to produce a superset of groups. The aggregate function specified in the select list is applied to these groups to produce summary values for the additional superaggregate rows. The number of extra groups in the results set is determined by the number of columns included in the GROUP BY clause.

In fact, every possible combination of the columns or expressions in the GROUP BY clause is used to produce superaggregates. If you have n columns or expressions in the GROUP BY clause, there will be 2^n possible superaggregate combinations. Mathematically, these combinations form an n -dimensional cube, which is how the operator got its name.

By using application or programming tools, these superaggregate values can then be fed into charts and graphs that convey results and relationships visually and effectively.

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.

ORACLE

3-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of a CUBE Operator

The output of the SELECT statement in the example can be interpreted as follows:

- The total salary for every job within a department (for those departments whose department ID is less than 60) is displayed by the GROUP BY clause (labeled 1)
- The total salary for those departments whose department ID is less than 60 (labeled 2)
- The total salary for every job irrespective of the department (labeled 3)
- Total salary for those departments whose department ID is less than 60, irrespective of the job titles (labeled 4)

In the preceding example, all rows indicated as 1 are regular rows, all rows indicated as 2 and 4 are superaggregate rows, and all rows indicated as 3 are cross-tabulation values.

The CUBE operator has also performed the ROLLUP operation to display the subtotals for those departments whose department ID is less than 60 and the total salary for those departments whose department ID is less than 60, irrespective of the job titles. Additionally, the CUBE operator displays the total salary for every job irrespective of the department.

Note: Similar to the ROLLUP operator, producing subtotals in n dimensions (that is, n columns in the GROUP BY clause) without a CUBE operator requires 2^n SELECT statements to be linked with UNION ALL. Thus, a report with three dimensions requires $2^3 = 8$ SELECT statements to be linked with UNION ALL.

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

ORACLE

3-11

Copyright © Oracle Corporation, 2001. All rights reserved.

The GROUPING Function

The GROUPING function can be used with either the CUBE or ROLLUP operator to help you understand how a summary value has been obtained.

The GROUPING function uses a single column as its argument. The *expr* in the GROUPING function must match one of the expressions in the GROUP BY clause. The function returns a value of 0 or 1.

The values returned by the GROUPING function are useful to:

- Determine the level of aggregation of a given subtotal; that is, the group or groups on which the subtotal is based
- Identify whether a NULL value in the expression column of a row of the result set indicates:
 - A NULL value from the base table (stored NULL value)
 - A NULL value created by ROLLUP/CUBE (as a result of a group function on that expression)

A value of 0 returned by the GROUPING function based on an expression indicates one of the following:

- The expression has been used to calculate the aggregate value.
- The NULL value in the expression column is a stored NULL value.

A value of 1 returned by the GROUPING function based on an expression indicates one of the following:

- The expression has not been used to calculate the aggregate value.
- The NULL value in the expression column is created by ROLLUP or CUBE as a result of grouping.

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.

ORACLE

Example of a GROUPING Function

In the example in the slide, consider the summary value 4400 in the first row (labeled 1). This summary value is the total salary for the job ID of AD_ASST within department 10. To calculate this summary value, both the columns DEPARTMENT_ID and JOB_ID have been taken into account. Thus a value of 0 is returned for both the expressions GROUPING(department_id) and GROUPING(job_id).

Consider the summary value 4400 in the second row (labeled 2). This value is the total salary for department 10 and has been calculated by taking into account the column DEPARTMENT_ID; thus a value of 0 has been returned by GROUPING(department_id). Because the column JOB_ID has not been taken into account to calculate this value, a value of 1 has been returned for GROUPING(job_id). You can observe similar output in the fifth row.

In the last row, consider the summary value 23400 (labeled 3). This is the total salary for those departments whose department ID is less than 50 and all job titles. To calculate this summary value, neither of the columns DEPARTMENT_ID and JOB_ID have been taken into account. Thus a value of 1 is returned for both the expressions GROUPING(department_id) and GROUPING(job_id).

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 greater the performance benefit.

ORACLE

3-13

Copyright © Oracle Corporation, 2001. All rights reserved.

GROUPING SETS

GROUPING SETS are a further extension of the GROUP BY clause that let you specify multiple groupings of data. Doing so facilitates efficient aggregation and hence facilitates analysis of data across multiple dimensions.

A single SELECT statement can now be written using GROUPING SETS to specify various groupings (that can also include ROLLUP or CUBE operators), rather than multiple SELECT statements combined by UNION ALL operators. For example, you can say:

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

This statement calculates aggregates over three groupings:

```
(department_id, job_id, manager_id), (department_id, manager_id)
and (job_id, manager_id)
```

Without this enhancement in Oracle9i, multiple queries combined together with UNION ALL are required to get the output of the preceding SELECT statement. A multiquery approach is inefficient, for it requires multiple scans of the same data.

GROUPING SETS (continued)

Compare the preceding statement with this alternative:

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

The preceding statement computes all the 8 (2 *2 *2) groupings, though only the groups (department_id, job_id, manager_id), (department_id, manager_id) and (job_id, manager_id) are of interest to you.

Another alternative is the following statement:

```
SELECT department_id, job_id, manager_id, AVG(salary)
FROM employees
GROUP BY department_id, job_id, manager_id
UNION ALL
SELECT department_id, NULL, manager_id, AVG(salary)
FROM employees
GROUP BY department_id, manager_id
UNION ALL
SELECT NULL, job_id, manager_id, AVG(salary)
FROM employees
GROUP BY job_id, manager_id;
```

This statement requires three scans of the base table, making it inefficient.

CUBE and ROLLUP can be thought of as grouping sets with very specific semantics. The following equivalencies show this fact:

CUBE(a, b, c) is equivalent to	GROUPING SETS ((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ())
ROLLUP(a, b, c) is equivalent to	GROUPING SETS ((a, b, c), (a, b), (a), ())

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
...			
	SA_MAN	100	10500
	SA_REP	149	8866.66667
	ST_CLERK	124	2925
	ST_MAN	100	5800

26 rows selected.

1

2

ORACLE

3-15

Copyright © Oracle Corporation, 2001. All rights reserved.

GROUPING SETS: Example

The query in the slide calculates aggregates over two groupings. The table is divided into the following groups:

- Department ID, Job ID
- Job ID, Manager ID

The average salaries for each of these groups are calculated. The results set displays average salary for each of the two groups.

In the output, the group marked as 1 can be interpreted as:

- The average salary of all employees with the job ID AD_ASST in the department 10 is 4400.
- The average salary of all employees with the job ID MK_MAN in the department 20 is 13000.
- The average salary of all employees with the job ID MK_REP in the department 20 is 6000.
- The average salary of all employees with the job ID ST_CLERK in the department 50 is 2925 and so on.

GROUPING SETS: Example (continued)

The group marked as 2 in the output is interpreted as:

- The average salary of all employees with the job ID MK_REP, who report to the manager with the manager ID 201, is 6000.
- The average salary of all employees with the job ID SA_MAN, who report to the manager with the manager ID 100, is 10500, and so on.

The example in the slide can also be written as:

```
SELECT department_id, job_id, NULL as manager_id,  
       AVG(salary) as AVGSAL  
FROM employees  
GROUP BY department_id, job_id  
UNION ALL  
SELECT NULL, job_id, manager_id, avg(salary) as AVGSAL  
FROM employees  
GROUP BY job_id, manager_id;
```

In the absence of an optimizer that looks across query blocks to generate the execution plan, the preceding query would need two scans of the base table, EMPLOYEES. This could be very inefficient. Hence the usage of the GROUPING SETS statement is recommended.

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, use the GROUP BY clause to 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.**

ORACLE

3-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Composite Columns

A composite column is a collection of columns that are treated as a unit during the computation of groupings. You specify the columns in parentheses as in the following statement:

```
ROLLUP (a, (b, c), d)
```

Here, (b, c) form a composite column and are treated as a unit. In general, composite columns are useful in ROLLUP, CUBE, and GROUPING SETS. For example, in CUBE or ROLLUP, composite columns would mean skipping aggregation across certain levels.

That is, GROUP BY ROLLUP(a, (b, c))

is equivalent to

```
GROUP BY a, b, c UNION ALL
GROUP BY a UNION ALL
GROUP BY ()
```

Here, (b, c) are treated as a unit and rollup will not be applied across (b, c). It is as if you have an alias, for example z, for (b, c), and the GROUP BY expression reduces to GROUP BY ROLLUP(a, z).

Note: GROUP BY () is typically a SELECT statement with NULL values for the columns a and b and only the aggregate function. This is generally used for generating the grand totals.

```
SELECT NULL, NULL, aggregate_col
FROM <table_name>
GROUP BY ( );
```

Composite Columns (continued)

Compare this with the normal ROLLUP as in:

```
GROUP BY ROLLUP(a, b, c)
```

which would be

```
GROUP BY a, b, c UNION ALL
```

```
GROUP BY a, b UNION ALL
```

```
GROUP BY a UNION ALL
```

```
GROUP BY ().
```

Similarly,

```
GROUP BY CUBE((a, b), c)
```

would be equivalent to

```
GROUP BY a, b, c UNION ALL
```

```
GROUP BY a, b UNION ALL
```

```
GROUP BY c UNION ALL
```

```
GROUP BY ().
```

The following table shows grouping sets specification and equivalent GROUP BY specification.

GROUPING SETS Statements	Equivalent GROUP BY Statements
GROUP BY GROUPING SETS(a, b, c)	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY c
GROUP BY GROUPING SETS(a, b, (b, c)) (The GROUPING SETS expression has a composite column)	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY b, c
GROUP BY GROUPING SETS((a, b, c))	GROUP BY a, b, c
GROUP BY GROUPING SETS(a, (b), ())	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY ()
GROUP BY GROUPING SETS (a, ROLLUP(b, c)) (The GROUPING SETS expression has a composite column)	GROUP BY a UNION ALL GROUP BY ROLLUP(b, c)

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
...			
			175500

23 rows selected.

ORACLE

3-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Composite Columns: Example

Consider the example:

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

The preceding query results in the Oracle Server computing the following groupings:

1. (department_id, job_id, manager_id)
2. (department_id, job_id)
3. (department_id)
4. ()

If you are just interested in grouping of lines (1), (3), and (4) in the preceding example, you cannot limit the calculation to those groupings without using composite columns. With composite columns, this is possible by treating JOB_ID and MANAGER_ID columns as a single unit while rolling up. Columns enclosed in parentheses are treated as a unit while computing ROLLUP and CUBE. This is illustrated in the example on the slide. By enclosing JOB_ID and MANAGER_ID columns in parenthesis, we indicate to the Oracle Server to treat JOB_ID and MANAGER_ID as a single unit, as a composite column.

Composite Columns Example (continued)

The example in the slide computes the following groupings:

- (department_id, job_id, manager_id)
- (department_id)
- ()

The example in the slide displays the following:

- Total salary for every department (labeled 1)
- Total salary for every department, job ID, and manager (labeled 2)
- Grand total (labeled 3)

The example in the slide can also be written as:

```
SELECT department_id, job_id, manager_id, SUM(salary)
FROM   employees
GROUP BY department_id, job_id, manager_id
UNION ALL
SELECT department_id, TO_CHAR(NULL), TO_NUMBER(NULL), SUM(salary)
FROM   employees
GROUP BY department_id
UNION ALL
SELECT TO_NUMBER(NULL), TO_CHAR(NULL), TO_NUMBER(NULL), SUM(salary)
FROM   employees
GROUP BY ( );
```

In the absence of an optimizer that looks across query blocks to generate the execution plan, the preceding query would need three scans of the base table, EMPLOYEES. This could be very inefficient. Hence, the use of composite columns is recommended.

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)
```

ORACLE

3-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Concatenated Columns

Concatenated groupings offer a concise way to generate useful combinations of groupings. The concatenated groupings are specified simply by listing multiple grouping sets, cubes, and rollups, and separating them with commas. Here is an example of concatenated grouping sets:

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

The preceding SQL defines the following groupings:

(a, c), (a, d), (b, c), (b, d)

Concatenation of grouping sets is very helpful for these reasons:

- **Ease of query development:** you need not manually enumerate all groupings
- **Use by applications:** SQL generated by OLAP applications often involves concatenation of grouping sets, with each grouping set defining groupings needed for a dimension

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)
1	10	AD_ASST	101	4400
	20	MK_MAN	100	13000
	...			
2	10		101	4400
	20		100	13000
	...			
3	10	AD_ASST		4400
	10			4400
	...			
4		SA_REP		7000
				7000

49 rows selected.

ORACLE

3-22

Copyright © Oracle Corporation, 2001. All rights reserved.

Concatenated Groupings Example

The example in the slide results in the following groupings:

- (department_id, manager_id, job_id)
- (department_id, manager_id)
- (department_id, job_id)
- (department_id)

The total salary for each of these groups is calculated.

The example in the slide displays the following:

- Total salary for every department, job ID, manager (labeled 1)
- Total salary for every department, manager ID(labeled 2)
- Total salary for every department, job ID(labeled 3)
- Total salary for every department (labeled 4)

For easier understanding, the details for the department 10 are highlighted in the output.

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

ORACLE

3-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

- **ROLLUP** and **CUBE** are extensions of the **GROUP BY** clause.
- **ROLLUP** is used to display subtotal and grand total values.
- **CUBE** is used to display cross-tabulation values.
- The **GROUPING** function helps you determine whether a row is an aggregate produced by a **CUBE** or **ROLLUP** operator.
- With the **GROUPING SETS** syntax, you can define multiple groupings in the same query. **GROUP BY** computes all the groupings specified and combines them with **UNION ALL**.
- Within the **GROUP BY** clause, you can combine expressions in various ways:
 - To specify composite columns, you group columns within parentheses so that the Oracle Server treats them as a unit while computing **ROLLUP** or **CUBE** operations.
 - 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.

Practice 3 Overview

This practice covers the following topics:

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

ORACLE

3-24

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 3 Overview

In this practice, you use the `ROLLUP` and `CUBE` operators as extensions of the `GROUP BY` clause. You will also use `GROUPING SETS`.

Practice 3

1. Write a query to display the following for those employees whose manager ID is less than 120:
 - Manager ID
 - Job ID and total salary for every job ID for employees who report to the same manager
 - Total salary of those managers
 - Total salary of those managers, irrespective of the job IDs

MANAGER_ID	JOB_ID	SUM(SALARY)
100	AD_VP	34000
100	MK_MAN	13000
100	SA_MAN	10500
100	ST_MAN	5800
100		63300
101	AC_MGR	12000
101	AD_ASST	4400
101		16400
102	IT_PROG	9000
102		9000
103	IT_PROG	10200
103		10200
		98900

13 rows selected.

Practice 3 (continued)

2. Observe the output from question 1. Write a query using the GROUPING function to determine whether the NULL values in the columns corresponding to the GROUP BY expressions are caused by the ROLLUP operation.

MGR	JOB	SUM(SALARY)	GROUPING(MANAGER_ID)	GROUPING(JOB_ID)
100	AD_VP	34000	0	0
100	MK_MAN	13000	0	0
100	SA_MAN	10500	0	0
100	ST_MAN	5800	0	0
100		63300	0	1
101	AC_MGR	12000	0	0
101	AD_ASST	4400	0	0
101		16400	0	1
102	IT_PROG	9000	0	0
102		9000	0	1
103	IT_PROG	10200	0	0
103		10200	0	1
		98900	1	1

13 rows selected.

Practice 3 (continued)

3. Write a query to display the following for those employees whose manager ID is less than 120:
- Manager ID
 - Job and total salaries for every job for employees who report to the same manager
 - Total salary of those managers
 - Cross-tabulation values to display the total salary for every job, irrespective of the manager

MANAGER_ID	JOB_ID	SUM(SALARY)
100	AD_VP	34000
100	MK_MAN	13000
100	SA_MAN	10500
100	ST_MAN	5800
100		63300
101	AC_MGR	12000
101	AD_ASST	4400
101		16400
102	IT_PROG	9000
102		9000
103	IT_PROG	10200
103		10200
	AC_MGR	12000
	AD_ASST	4400
MANAGER_ID	JOB_ID	SUM(SALARY)
	AD_VP	34000
	IT_PROG	19200
	MK_MAN	13000
	SA_MAN	10500
	ST_MAN	5800
		98900

20 rows selected.

Practice 3 (continued)

- Observe the output from question 3. Write a query using the GROUPING function to determine whether the NULL values in the columns corresponding to the GROUP BY expressions are caused by the CUBE operation.

MGR	JOB	SUM(SALARY)	GROUPING(MANAGER_ID)	GROUPING(JOB_ID)
100	AD_VP	34000	0	0
100	MK_MAN	13000	0	0
100	SA_MAN	10500	0	0
100	ST_MAN	5800	0	0
100		63300	0	1
101	AC_MGR	12000	0	0
101	AD_ASST	4400	0	0
101		16400	0	1
102	IT_PROG	9000	0	0
102		9000	0	1
103	IT_PROG	10200	0	0
103		10200	0	1
	AC_MGR	12000	1	0
	AD_ASST	4400	1	0
MGR	JOB	SUM(SALARY)	GROUPING(MANAGER_ID)	GROUPING(JOB_ID)
	AD_VP	34000	1	0
	IT_PROG	19200	1	0
	MK_MAN	13000	1	0
	SA_MAN	10500	1	0
	ST_MAN	5800	1	0
		98900	1	1

20 rows selected.

Practice 3 (continued)

5. Using GROUPING SETS, write a query to display the following groupings:

- department_id, manager_id, job_id
- department_id, job_id
- manager_id, job_id

The query should calculate the sum of the salaries for each of these groups.

DEPARTMENT_ID	MANAGER_ID	JOB_ID	SUM(SALARY)
10	101	AD_ASST	4400
20	100	MK_MAN	13000
20	201	MK_REP	6000
50	124	ST_CLERK	11700
50	100	ST_MAN	5800
60	102	IT_PROG	9000
60	103	IT_PROG	10200
80	100	SA_MAN	10500
80	149	SA_REP	19600
90		AD PRES	24000
90	100	AD_VP	34000
110	205	AC_ACCOUNT	8300
110	101	AC_MGR	12000
	149	SA_REP	7000
...			
	100	MK_MAN	13000
	100	SA_MAN	10500
	100	ST_MAN	5800
	101	AC_MGR	12000
	101	AD_ASST	4400
	102	IT_PROG	9000
	103	IT_PROG	10200
	124	ST_CLERK	11700
	149	SA_REP	26600
	201	MK_REP	6000
	205	AC_ACCOUNT	8300
		AD PRES	24000

40 rows selected.

4

Advanced Subqueries

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

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

ORACLE

4-2

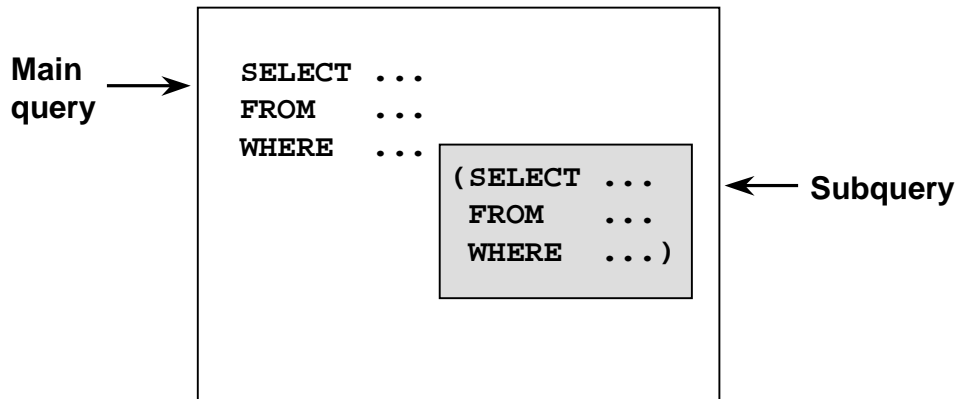
Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

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

What Is a Subquery?

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



ORACLE

What Is a Subquery?

A *subquery* is a `SELECT` statement that is embedded in a clause of another SQL statement, called the parent statement.

The subquery (inner query) returns a value that is used by the parent statement. Using a nested subquery is equivalent to performing two sequential queries and using the result of the inner query as the search value in the outer query (main query).

Subqueries can be used for the following purposes:

- To provide values for conditions in `WHERE`, `HAVING`, and `START WITH` clauses of `SELECT` statements
- To define the set of rows to be inserted into the target table of an `INSERT` or `CREATE TABLE` statement
- To define the set of rows to be included in a view or snapshot in a `CREATE VIEW` or `CREATE SNAPSHOT` statement
- To define one or more values to be assigned to existing rows in an `UPDATE` statement
- To define a table to be operated on by a containing query. (You do this by placing the subquery in the `FROM` clause. This can be done in `INSERT`, `UPDATE`, and `DELETE` statements as well.)

Note: A subquery is evaluated once for the entire parent statement.

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).

ORACLE

4-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Subqueries

You can build powerful statements out of simple ones by using subqueries. Subqueries can be very useful when you need to select rows from a table with a condition that depends on the data in the table itself or some other table. Subqueries are very useful for writing SQL statements that need values based on one or more unknown conditional values.

In the syntax:

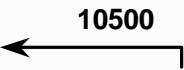
operator includes a comparison operator such as >, =, or IN

Note: Comparison operators fall into two classes: single-row operators (>, =, >=, <, <>, <=) and multiple-row operators (IN, ANY, ALL).

The subquery is often referred to as a nested SELECT, sub-SELECT, or inner SELECT statement. The inner and outer queries can retrieve data from either the same table or different tables.

Using a Subquery

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



LAST_NAME
King
Kochhar
De Haan
Abel
Hartstein
Higgins

6 rows selected.

ORACLE

Using a Subquery

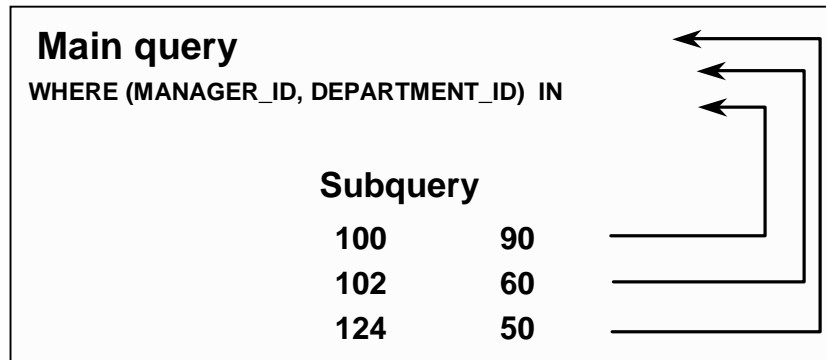
In the example in the slide, the inner query returns the salary of the employee with employee number 149. The outer query uses the result of the inner query to display the names of all the employees who earn more than this amount.

Example

Display the names of all employees who earn less than the average salary in the company.

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  salary < (SELECT AVG(salary)
                 FROM   employees);
```

Multiple-Column Subqueries



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

ORACLE

4-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Multiple-Column Subqueries

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

Syntax

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

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

Column Comparisons

Column comparisons in a multiple-column subquery can be:

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

ORACLE

4-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Pairwise versus Nonpairwise Comparisons

Column comparisons in a multiple-column subquery can be pairwise comparisons or nonpairwise comparisons.

In the example on the next slide, a pairwise comparison was executed in the WHERE clause. Each candidate row in the SELECT statement must have *both* the same MANAGER_ID column and the DEPARTMENT_ID as the employee with the EMPLOYEE_ID 178 or 174.

A multiple-column subquery can also be a nonpairwise comparison. In a nonpairwise comparison, each of the columns from the WHERE clause of the parent SELECT statement are individually compared to multiple values retrieved by the inner select statement. The individual columns can match any of the values retrieved by the inner select statement. But collectively, all the multiple conditions of the main SELECT statement must be satisfied for the row to be displayed. The example on the next page illustrates a nonpairwise comparison.

Pairwise Comparison Subquery

Display the details of the employees who are managed by the same manager *and* work in the same department as the employees with EMPLOYEE_ID 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);
```

ORACLE

4-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Pairwise Comparison Subquery

The example in the slide is that of a multiple-column subquery because the subquery returns more than one column. It compares the values in the MANAGER_ID column and the DEPARTMENT_ID column of each row in the EMPLOYEES table with the values in the MANAGER_ID column and the DEPARTMENT_ID column for the employees with the EMPLOYEE_ID 178 or 174.

First, the subquery to retrieve the MANAGER_ID and DEPARTMENT_ID values for the employees with the EMPLOYEE_ID 178 or 174 is executed. These values are compared with the MANAGER_ID column and the DEPARTMENT_ID column of each row in the EMPLOYEES table. If the values match, the row is displayed. In the output, the records of the employees with the EMPLOYEE_ID 178 or 174 will not be displayed. The output of the query in the slide follows.

EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
176	149	80

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);
```

ORACLE

4-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Nonpairwise Comparison Subquery

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

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

EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
142	124	50
143	124	50
144	124	50
176	149	80

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.

ORACLE

Using a Subquery in the FROM Clause

You can use a subquery in the FROM clause of a SELECT statement, which is very similar to how views are used. A subquery in the FROM clause of a SELECT statement is also called an *inline* view. A subquery in the FROM clause of a SELECT statement defines a data source for that particular SELECT statement, and only that SELECT statement. The example on the slide displays employee last names, salaries, department numbers, and average salaries for all the employees who earn more than the average salary in their department. The subquery in the FROM clause is named *b*, and the outer query references the SALAVG column using this alias.

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` and `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`

ORACLE

4-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Scalar Subqueries in SQL

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

The value of the scalar subquery expression is the value of the select list item of the subquery. If the subquery returns 0 rows, the value of the scalar subquery expression is `NULL`. If the subquery returns more than one row, the Oracle Server returns an error. The Oracle Server has always supported the usage of a scalar subquery in a `SELECT` statement. The usage of scalar subqueries has been enhanced in Oracle9i. You can now use scalar subqueries in:

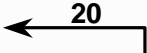
- Condition and expression part of `DECODE` and `CASE`
- All clauses of `SELECT` except `GROUP BY`
- In the left-hand side of the operator in the `SET` clause and `WHERE` clause of `UPDATE` statement

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

- As default values for columns and hash expressions for clusters
- In the `RETURNING` clause of DML statements
- As the basis of a function-based index
- In `GROUP BY` clauses, `CHECK` constraints, `WHEN` conditions
- `HAVING` clauses
- In `START WITH` and `CONNECT BY` clauses
- In statements that are unrelated to queries, such as `CREATE PROFILE`

Scalar Subqueries: Examples

Scalar Subqueries in CASE Expressions

```
SELECT employee_id, last_name,  
       (CASE  
         WHEN department_id =   
           (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);
```

ORACLE

4-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Scalar Subqueries: Examples

The first example in the slide demonstrates that scalar subqueries can be used in CASE expressions. The inner query returns the value 20, which is the department ID of the department whose location ID is 1800. The CASE expression in the outer query uses the result of the inner query to display the employee ID, last names, and a value of Canada or USA, depending on whether the department ID of the record retrieved by the outer query is 20 or not.

The result of the preceding example follows:

EMPLOYEE_ID	LAST_NAME	LOCATI
100	King	USA
101	Kochhar	USA
102	De Haan	USA
...		
201	Hartstein	Canada
202	Fay	Canada
205	Higgins	USA
206	Gietz	USA

20 rows selected.

Scalar Subqueries: Examples (Continued)

The second example in the slide demonstrates that scalar subqueries can be used in the ORDER BY clause. The example orders the output based on the DEPARTMENT_NAME by matching the DEPARTMENT_ID from the EMPLOYEES table with the DEPARTMENT_ID from the DEPARTMENTS table. This comparison is done in a scalar subquery in the ORDER BY clause. The result of the second example follows:

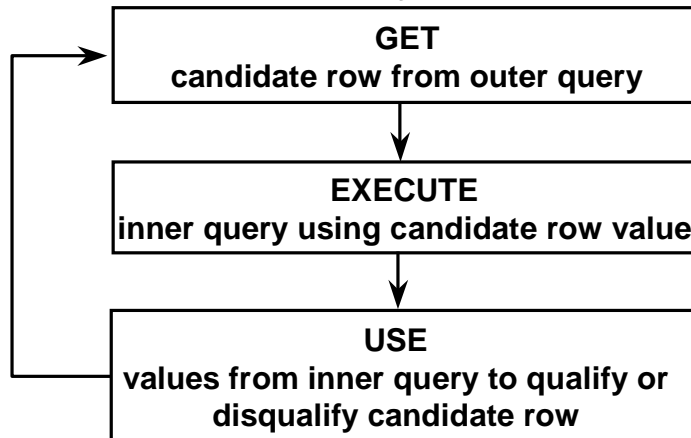
EMPLOYEE_ID	LAST_NAME
205	Higgins
206	Gietz
200	Whalen
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst
107	Lorentz
201	Hartstein
202	Fay
149	Zlotkey
176	Taylor
174	Abel
EMPLOYEE_ID	LAST_NAME
124	Mourgos
141	Rajs
142	Davies
143	Matos
144	Vargas
178	Grant

20 rows selected.

The second example uses a correlated subquery. In a correlated subquery, the subquery references a column from a table referred to in the parent statement. Correlated subqueries are explained later in this lesson.

Correlated Subqueries

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



ORACLE

Correlated Subqueries

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

Nested Subqueries Versus Correlated Subqueries

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

Nested Subquery Execution

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

Correlated Subquery Execution

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

Correlated Subqueries

```
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.

ORACLE

Correlated Subqueries (continued)

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


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

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

Using Correlated Subqueries

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) ;
```



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

ORACLE

4-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Using Correlated Subqueries

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

Because both the outer query and inner query use the EMPLOYEES table in the FROM clause, an alias is given to EMPLOYEES in the outer SELECT statement, for clarity. Not only does the alias make the entire SELECT statement more readable, but without the alias the query would not work properly, because the inner statement would not be able to distinguish the inner table column from the outer table column.

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

ORACLE

Using Correlated Subqueries (continued)

The example in the slide displays the details of those employees who have switched jobs at least twice. The Oracle Server evaluates a correlated subquery as follows:

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

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

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

ORACLE

The EXISTS Operator

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

Using the EXISTS Operator

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.

ORACLE

Using the EXISTS Operator

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

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

Note that the inner SELECT query does not need to return a specific value, so a constant can be selected. From a performance standpoint, it is faster to select a constant than a column.

Note: Having EMPLOYEE_ID in the SELECT clause of the inner query causes a table scan for that column. Replacing it with the literal X, or any constant, improves performance. This is more efficient than using the IN operator.

A IN construct can be used as an alternative for a EXISTS operator, as shown in the following example:

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees
WHERE  employee_id IN (SELECT manager_id
                      FROM   employees
                      WHERE  manager_id IS NOT NULL);
```

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

ORACLE

Using the NOT EXISTS Operator

Alternative Solution

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

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

no rows selected

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

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.

ORACLE

Correlated UPDATE

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

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);
```

ORACLE

4-22

Copyright © Oracle Corporation, 2001. All rights reserved.

Correlated UPDATE (continued)

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

Here is another example for a correlated update.

Problem Statement

Use a correlated subquery to update rows in the **EMPLOYEES** table based on rows from the **REWARDS** table:

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


Correlated UPDATE (continued)

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

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

Correlated DELETE

```
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.

ORACLE[®]

4-24

Copyright © Oracle Corporation, 2001. All rights reserved.

Correlated DELETE

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

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

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);
```

ORACLE

4-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Correlated DELETE (continued)

Example

Two tables are used in this example. They are:

- The **EMPLOYEES** table, which gives details of all the current employees
- The **EMP_HISTORY** table, which gives details of previous employees

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

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

ORACLE

4-26

Copyright © Oracle Corporation, 2001. All rights reserved.

The WITH Clause

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

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

WITH Clause Benefits

- Makes the query easy to read
- Evaluates a clause only once, even if it appears multiple times in the query, thereby enhancing 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.

ORACLE

4-27

Copyright © Oracle Corporation, 2001. All rights reserved.

WITH Clause: Example

The problem in the slide would require the following intermediate calculations:

1. Calculate the total salary for every department, and store the result using a WITH clause.
2. Calculate the average salary across departments, and store the result using a WITH clause.
3. Compare the total salary calculated in the first step with the average salary calculated in the second step. If the total salary for a particular department is greater than the average salary across departments, display the department name and the total salary for that department.

The solution for the preceding problem is given in the next page.

WITH Clause: Example

WITH

```
dept_costs AS (  
    SELECT d.department_name, SUM(e.salary) AS dept_total  
    FROM   employees e, departments d  
    WHERE  e.department_id = d.department_id  
    GROUP BY d.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 avg_cost)  
ORDER BY department_name;
```

ORACLE

4-28

Copyright © Oracle Corporation, 2001. All rights reserved.

WITH Clause: Example (continued)

The SQL code in the slide is an example of a situation in which you can improve performance and write SQL more simply by using the WITH clause. The query creates the query names DEPT_COSTS and AVG_COST and then uses them in the body of the main query. Internally, the WITH clause is resolved either as an in-line view or a temporary table. The optimizer chooses the appropriate resolution depending on the cost or benefit of temporarily storing the results of the WITH clause.

Note: A subquery in the FROM clause of a SELECT statement is also called an in-line view.

The output generated by the SQL code on the slide will be as follows:

DEPARTMENT_NAME	DEPT_TOTAL
Executive	58000
Sales	30100

The WITH Clause Usage Notes

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

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 Oracle9i.**

ORACLE

4-29

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

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

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

Oracle9i enhances the uses of scalar subqueries. Scalar subqueries can now be used in:

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

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**

ORACLE

Summary

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

Practice 4 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**

ORACLE

4-31

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 4 Overview

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

Practice 4

1. Write a query to display the last name, department number, and salary of any employee whose department number and salary both match the department number and salary of any employee who earns a commission.

LAST_NAME	DEPARTMENT_ID	SALARY
Taylor	80	8600
Zlotkey	80	10500
Abel	80	11000

2. Display the last name, department name, and salary of any employee whose salary and commission match the salary and commission of any employee located in location ID 1700.

LAST_NAME	DEPARTMENT_NAME	SALARY
Whalen	Administration	4400
Gietz	Accounting	8300
Higgins	Accounting	12000
Kochhar	Executive	17000
De Haan	Executive	17000
King	Executive	24000

6 rows selected.

3. Create a query to display the last name, hire date, and salary for all employees who have the same salary and commission as Kochhar.

Note: Do not display Kochhar in the result set.

LAST_NAME	HIRE_DATE	SALARY
De Haan	13-JAN-93	17000

4. Create a query to display the employees who earn a salary that is higher than the salary of all of the sales managers (JOB_ID = 'SA_MAN'). Sort the results on salary from highest to lowest.

LAST_NAME	JOB_ID	SALARY
King	AD_PRES	24000
Kochhar	AD_VP	17000
De Haan	AD_VP	17000
Hartstein	MK_MAN	13000
Higgins	AC_MGR	12000
Abel	SA_REP	11000

6 rows selected.

Practice 4 (continued)

5. Display the details of the employee ID, last name, and department ID of those employees who live in cities whose name begins with *T*.

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
201	Hartstein	20
202	Fay	20

6. Write a query to find all employees who earn more than the average salary in their departments. Display last name, salary, department ID, and the average salary for the department. Sort by average salary. Use aliases for the columns retrieved by the query as shown in the sample output.

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

7 rows selected.

7. Find all employees who are not supervisors.
a. First do this using the NOT EXISTS operator.

LAST_NAME
Ernst
Lorentz
Rajs
Davies
Matos
Vargas
Abel
Taylor
Grant
Whalen
Fay
Gietz

12 rows selected.

- b. Can this be done by using the NOT IN operator? How, or why not?

Practice 4 (continued)

8. Write a query to display the last names of the employees who earn less than the average salary in their departments.

LAST_NAME
Kochhar
De Haan
Ernst
Lorentz
Davies
Matos
Vargas
Taylor
Fay
Gietz

10 rows selected.

9. Write a query to display the last names of the employees who have one or more coworkers in their departments with later hire dates but higher salaries.

LAST_NAME
Rajs
Davies
Matos
Vargas
Taylor

Practice 4 (continued)

10. Write a query to display the employee ID, last names, and department names of all employees.

Note: Use a scalar subquery to retrieve the department name in the SELECT statement.

EMPLOYEE_ID	LAST_NAME	DEPARTMENT
205	Higgins	Accounting
206	Gietz	Accounting
200	Whalen	Administration
100	King	Executive
101	Kochhar	Executive
102	De Haan	Executive
103	Hunold	IT
104	Ernst	IT
107	Lorentz	IT
201	Hartstein	Marketing
202	Fay	Marketing
149	Zlotkey	Sales
176	Taylor	Sales
174	Abel	Sales
EMPLOYEE_ID	LAST_NAME	DEPARTMENT
124	Mourgos	Shipping
141	Rajs	Shipping
142	Davies	Shipping
143	Matos	Shipping
144	Vargas	Shipping
178	Grant	

20 rows selected.

11. Write a query to display the department names of those departments whose total salary cost is above one eighth (1/8) of the total salary cost of the whole company. Use the WITH clause to write this query. Name the query SUMMARY.

DEPARTMENT_NAME	DEPT_TOTAL
Executive	58000
Sales	30100

5

Hierarchical Retrieval

ORACLE[®]

Copyright © Oracle Corporation, 2001. All rights reserved.

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**

ORACLE[®]

5-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this lesson, you learn how to use hierarchical queries to create tree-structured reports.

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
EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
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

20 rows selected.

ORACLE

Sample Data from the EMPLOYEES Table

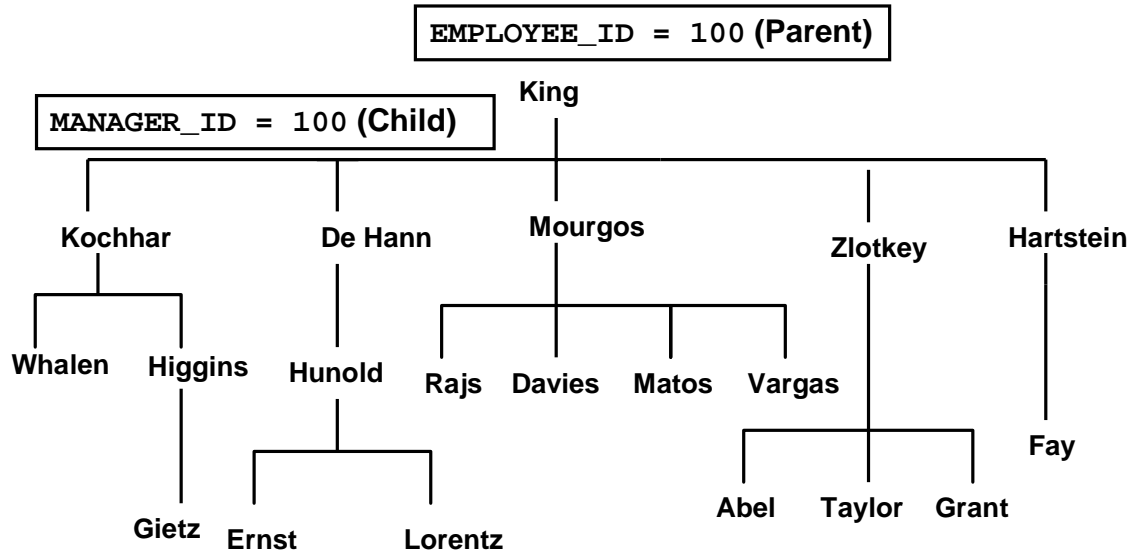
Using hierarchical queries, you can retrieve data based on a natural hierarchical relationship between rows in a table. A relational database does not store records in a hierarchical way. However, where a hierarchical relationship exists between the rows of a single table, a process called *tree walking* enables the hierarchy to be constructed. A hierarchical query is a method of reporting, in order, the branches of a tree.

Imagine a family tree with the eldest members of the family found close to the base or trunk of the tree and the youngest members representing branches of the tree. Branches can have their own branches, and so on.

A hierarchical query is possible when a relationship exists between rows in a table. For example, in the slide, you see that employees with the job IDs of AD_VP, ST_MAN, SA_MAN, and MK_MAN report directly to the president of the company. We know this because the MANAGER_ID column of these records contain the employee ID 100, which belongs to the president (AD_PRES).

Note: Hierarchical trees are used in various fields such as human genealogy (family trees), livestock (breeding purposes), corporate management (management hierarchies), manufacturing (product assembly), evolutionary research (species development), and scientific research.

Natural Tree Structure



ORACLE

5-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Natural Tree Structure

The EMPLOYEES table has a tree structure representing the management reporting line. The hierarchy can be created by looking at the relationship between equivalent values in the EMPLOYEE_ID and MANAGER_ID columns. This relationship can be exploited by joining the table to itself. The MANAGER_ID column contains the employee number of the employee's manager.

The parent-child relationship of a tree structure enables you to control:

- The direction in which the hierarchy is walked
- The starting point inside the hierarchy

Note: The slide displays an inverted tree structure of the management hierarchy of the employees in the EMPLOYEES table.

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
```

ORACLE

5-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Keywords and Clauses

Hierarchical queries can be identified by the presence of the `CONNECT BY` and `START WITH` clauses.

In the syntax:

<code>SELECT</code>	Is the standard <code>SELECT</code> clause.
<code>LEVEL</code>	For each row returned by a hierarchical query, the <code>LEVEL</code> pseudocolumn returns 1 for a root row, 2 for a child of a root, and so on.
<code>FROM table</code>	Specifies the table, view, or snapshot containing the columns. You can select from only one table.
<code>WHERE</code>	Restricts the rows returned by the query without affecting other rows of the hierarchy.
<i>condition</i>	Is a comparison with expressions.
<code>START WITH</code>	Specifies the root rows of the hierarchy (where to start). This clause is required for a true hierarchical query.
<code>CONNECT BY PRIOR</code>	Specifies the columns in which the relationship between parent and child rows exist. This clause is required for a hierarchical query.

The `SELECT` statement cannot contain a join or query from a view that contains a join.

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'
```

ORACLE

5-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Walking the Tree

The row or rows to be used as the root of the tree are determined by the **START WITH** clause. The **START WITH** clause can be used in conjunction with any valid condition.

Examples

Using the **EMPLOYEES** table, start with King, the president of the company.

```
... START WITH manager_id IS NULL
```

Using the **EMPLOYEES** table, start with employee Kochhar. A **START WITH** condition can contain a subquery.

```
... START WITH employee_id = (SELECT employee_id
                                FROM   employees
                                WHERE  last_name = 'Kochhar')
```

If the **START WITH** clause is omitted, the tree walk is started with all of the rows in the table as root rows. If a **WHERE** clause is used, the walk is started with all the rows that satisfy the **WHERE** condition. This no longer reflects a true hierarchy.

Note: The clauses **CONNECT BY PRIOR** and **START WITH** are not ANSI SQL standard.

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

ORACLE

5-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Walking the Tree (continued)

The direction of the query, whether it is from parent to child or from child to parent, is determined by the `CONNECT BY PRIOR` column placement. The `PRIOR` operator refers to the parent row. To find the children of a parent row, the Oracle Server evaluates the `PRIOR` expression for the parent row and the other expressions for each row in the table. Rows for which the condition is true are the children of the parent. The Oracle Server always selects children by evaluating the `CONNECT BY` condition with respect to a current parent row.

Examples

Walk from the top down using the `EMPLOYEES` table. Define a hierarchical relationship in which the `EMPLOYEE_ID` value of the parent row is equal to the `MANAGER_ID` value of the child row.

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

Walk from the bottom up using the `EMPLOYEES` table.

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

The `PRIOR` operator does not necessarily need to be coded immediately following the `CONNECT BY`. Thus, the following `CONNECT BY PRIOR` clause gives the same result as the one in the preceding example.

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

Note: The `CONNECT BY` clause cannot contain a subquery.

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	

ORACLE

5-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Walking the Tree: From the Bottom Up

The example in the slide displays a list of managers starting with the employee whose employee ID is 101.

Example

In the following example, EMPLOYEE_ID values are evaluated for the parent row and MANAGER_ID, and SALARY values are evaluated for the child rows. The PRIOR operator applies only to the EMPLOYEE_ID value.

```
... CONNECT BY PRIOR employee_id = manager_id
                AND salary > 15000;
```

To qualify as a child row, a row must have a MANAGER_ID value equal to the EMPLOYEE_ID value of the parent row and must have a SALARY value greater than \$15,000.

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	
Kochhar reports to King	
Whalen reports to Kochhar	
Higgins reports to Kochhar	
■ ■ ■	
Zlotkey reports to King	
Abel reports to Zlotkey	
Taylor reports to Zlotkey	
Grant reports to Zlotkey	
Hartstein reports to King	
Fay reports to Hartstein	

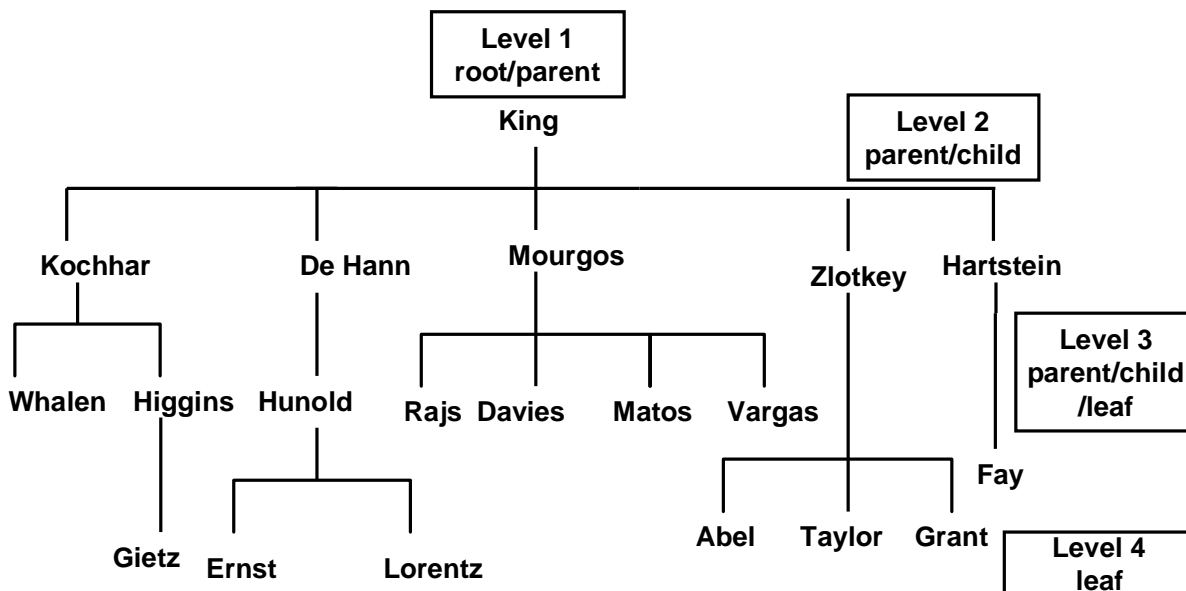
20 rows selected.

ORACLE

Walking the Tree: From the Top Down

Walking from the top down, display the names of the employees and their manager. Use employee King as the starting point. Print only one column.

Ranking Rows with the LEVEL Pseudocolumn



ORACLE

5-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Ranking Rows with the LEVEL Pseudocolumn

You can explicitly show the rank or level of a row in the hierarchy by using the LEVEL pseudocolumn. This will make your report more readable. The forks where one or more branches split away from a larger branch are called nodes, and the very end of a branch is called a leaf, or leaf node. The diagram in the slide shows the nodes of the inverted tree with their LEVEL values. For example, employee Higgins is a parent and a child, while employee Davies is a child and a leaf.

The LEVEL Pseudocolumn

Value	Level
1	A root node
2	A child of a root node
3	A child of a child, and so on

Note: A *root node* is the highest node within an inverted tree. A *child node* is any nonroot node. A *parent node* is any node that has children. A *leaf node* is any node without children. The number of levels returned by a hierarchical query may be limited by available user memory.

In the slide, King is the root or parent (LEVEL = 1). Kochhar, De Hann, Mourgos, Zlotkey, Hartstein, Higgins, and Hunold are children and also parents (LEVEL = 2). Whalen, Rajs, Davies, Matos, Vargas, Gietz, Ernst, Lorentz, Abel, Taylor, Grant, and Fay are children and leaves. (LEVEL = 3 and LEVEL = 4)

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
```

ORACLE

5-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Formatting Hierarchical Reports Using LEVEL

The nodes in a tree are assigned level numbers from the root. Use the LPAD function in conjunction with the pseudocolumn LEVEL to display a hierarchical report as an indented tree.

In the example on the slide:

- `LPAD(char1, n [, char2])` returns *char1*, left-padded to length *n* with the sequence of characters in *char2*. The argument *n* is the total length of the return value as it is displayed on your terminal screen.
- `LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-2, '_')` defines the display format.
- *char1* is the LAST_NAME, *n* the total length of the return value, is length of the LAST_NAME + (LEVEL*2) - 2, and *char2* is '_ '.

In other words, this tells SQL to take the LAST_NAME and left-pad it with the '_ ' character till the length of the resultant string is equal to the value determined by `LENGTH(last_name)+(LEVEL*2)-2`.

For King, LEVEL = 1. Hence, $(2 * 1) - 2 = 2 - 2 = 0$. So King does not get padded with any '_ ' character and is displayed in column 1.

For Kochhar, LEVEL = 2. Hence, $(2 * 2) - 2 = 4 - 2 = 2$. So Kochhar gets padded with 2 '_ ' characters and is displayed indented.

The rest of the records in the EMPLOYEES table are displayed similarly.

Formatting Hierarchical Reports Using LEVEL (continued)

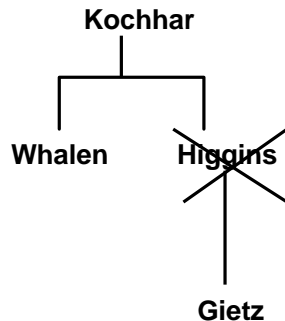
ORG_CHART	
King	
_Kochhar	
_Whalen	
_Higgins	
_Gietz	
_De Haan	
_Hunold	
_Ernst	
_Lorent z	
_Mourgos	
_Rajs	
_Davies	
_Matos	
_Vargas	
ORG_CHART	
_Zlotkey	
_Abel	
_Taylor	
_Grant	
_Hartstein	
_Fay	

20 rows selected.

Pruning Branches

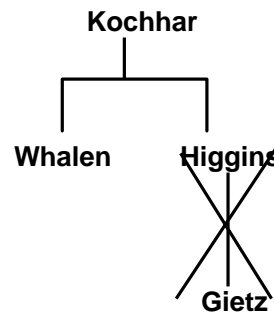
Use the WHERE clause
to eliminate a node.

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



Use the CONNECT BY clause
to eliminate a branch.

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



ORACLE

5-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Pruning Branches

You can use the WHERE and CONNECT BY clauses to prune the tree; that is, to control which nodes or rows are displayed. The predicate you use acts as a Boolean condition.

Examples

Starting at the root, walk from the top down, and eliminate employee Higgins in the result, but process the child rows.

```
SELECT department_id, employee_id, last_name, job_id, salary  
FROM employees  
WHERE last_name != 'Higgins'  
START WITH manager_id IS NULL  
CONNECT BY PRIOR employee_id = manager_id;
```

Starting at the root, walk from the top down, and eliminate employee Higgins and all child rows.

```
SELECT department_id, employee_id, last_name, job_id, salary  
FROM employees  
START WITH manager_id IS NULL  
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.**

ORACLE

5-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

You can use hierarchical queries to retrieve data based on a natural hierarchical relationship between rows in a table. The `LEVEL` pseudocolumn counts how far down a hierarchical tree you have traveled. You can specify the direction of the query using the `CONNECT BY PRIOR` clause. You can specify the starting point using the `START WITH` clause. You can use the `WHERE` and `CONNECT BY` clauses to prune the tree branches.

Practice 5 Overview

This practice covers the following topics:

- **Distinguishing hierarchical queries from nonhierarchical queries**
- **Walking through a tree**
- **Producing an indented report by using the `LEVEL` pseudocolumn**
- **Pruning the tree structure**
- **Sorting the output**

ORACLE

5-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 5 Overview

In this practice, you gain practical experience in producing hierarchical reports.

Paper-Based Questions

Question 1 is a paper-based question.

Practice 5

- Look at the following outputs. Are these outputs the result of a hierarchical query? Explain why or why not.

Exhibit 1:

EMPLOYEE_ID	LAST_NAME	MANAGER_ID	SALARY	DEPARTMENT_ID
100	King		24000	90
101	Kochhar	100	17000	90
102	De Haan	100	17000	90
201	Hartstein	100	13000	20
205	Higgins	101	12000	110
174	Abel	149	11000	80
149	Zlotkey	100	10500	80
103	Hunold	102	9000	60
■ ■ ■				
200	Whalen	101	4400	10
107	Lorentz	103	4200	60
141	Rajs	124	3500	50
142	Davies	124	3100	50
143	Matos	124	2600	50
144	Vargas	124	2500	50

20 rows selected.

Exhibit 2:

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
205	Higgins	110	Accounting
206	Gietz	110	Accounting
100	King	90	Executive
101	Kochhar	90	Executive
102	De Haan	90	Executive
149	Zlotkey	80	Sales
174	Abel	80	Sales
176	Taylor	80	Sales
103	Hunold	60	IT
104	Ernst	60	IT
107	Lorentz	60	IT

11 rows selected.

Practice 5 (continued)

Exhibit 3:

RANK	LAST_NAME
1	King
2	Kochhar
2	De Haan
3	Hunold
4	Ernst

2. Produce a report showing an organization chart for Mourgos's department. Print last names, salaries, and department IDs.

LAST_NAME	SALARY	DEPARTMENT_ID
Mourgos	5800	50
Rajs	3500	50
Davies	3100	50
Matos	2600	50
Vargas	2500	50

3. Create a report that shows the hierarchy of the managers for the employee Lorentz. Display his immediate manager first.

LAST_NAME
Hunold
De Haan
King

Practice 5 (continued)

4. Create an indented report showing the management hierarchy starting from the employee whose `LAST_NAME` is Kochhar. Print the employee's last name, manager ID, and department ID. Give alias names to the columns as shown in the sample output.

NAME	MGR	DEPTNO
Kochhar	100	90
__Whalen	101	10
__Higgins	101	110
___Gietz	205	110

If you have time, complete the following exercise:

5. Produce a company organization chart that shows the management hierarchy. Start with the person at the top level, exclude all people with a job ID of `IT_PROG`, and exclude De Haan and those employees who report to De Haan.

LAST_NAME	EMPLOYEE_ID	MANAGER_ID
King	100	
Kochhar	101	100
Whalen	200	101
Higgins	205	101
Gietz	206	205
Mourgos	124	100
Rajs	141	124
Davies	142	124
Matos	143	124
Vargas	144	124
Zlotkey	149	100
Abel	174	149
Taylor	176	149
Grant	178	149
LAST_NAME	EMPLOYEE_ID	MANAGER_ID
Hartstein	201	100
Fay	202	201

16 rows selected.



Oracle9i 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**

ORACLE

6-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

This lesson addresses the Oracle9i extensions to DDL and DML statements. It focuses on multitable INSERT statements, types of multitable INSERT statements, external tables, and the provision to 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 INSERT Statement

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

In the syntax:

<i>table</i>	is the name of the table
<i>column</i>	is the name of the column in the table to populate
<i>value</i>	is the corresponding value for the column

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

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.
```

ORACLE

Review of the UPDATE Statement

You can modify existing rows by using the UPDATE statement.

In the syntax:

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

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

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

ORACLE

6-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview of Multitable INSERT Statements

In a multitable **INSERT** statement, you insert computed rows derived from the rows returned from the evaluation of a subquery into one or more tables.

Multitable **INSERT** statements can play a very useful role in a data warehouse scenario. You need to load your data warehouse regularly so that it can serve its purpose of facilitating business analysis. To do this, data from one or more operational systems needs to be extracted and copied into the warehouse. The process of extracting data from the source system and bringing it into the data warehouse is commonly called ETL, which stands for extraction, transformation, and loading.

During extraction, the desired data has to be identified and extracted from many different sources, such as database systems and applications. After extraction, the data has to be physically transported to the target system or an intermediate system for further processing. Depending on the chosen means of transportation, some transformations can be done during this process. For example, a SQL statement that directly accesses a remote target through a gateway can concatenate two columns as part of the **SELECT** statement.

Once data is loaded into an Oracle9i database, data transformations can be executed using SQL operations. With Oracle9i multitable **INSERT** statements is one of the techniques for implementing SQL data transformations.

Overview of Multitable Insert Statements

Multitable `INSERT` statements offer the benefits of the `INSERT . . . SELECT` statement when multiple tables are involved as targets. Using functionality prior to Oracle9i, you had to deal with n independent `INSERT . . . SELECT` statements, thus processing the same source data n times and increasing the transformation workload n times.

As with the existing `INSERT . . . SELECT` statement, the new statement can be parallelized and used with the direct-load mechanism for faster performance.

Each record from any input stream, such as a nonrelational database table, can now be converted into multiple records for more relational database table environment. To implement this functionality before Oracle9i, you had to write multiple `INSERT` statements.

Types of Multitable INSERT Statements

Oracle9i introduces the following types of multitable insert statements:

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

ORACLE

6-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Types of Multitable INSERT Statements

Oracle9i introduces the following types of multitable INSERT statements:

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

You use different clauses to indicate the type of INSERT to be executed.

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]
```

ORACLE

6-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Multitable INSERT Statements

The slide displays the generic format for multitable INSERT statements. There are four types of multitable insert statements.

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

Unconditional INSERT: ALL into_clause

Specify ALL followed by multiple insert_into_clauses to perform an unconditional multitable insert. The Oracle Server executes each insert_into_clause once for each row returned by the subquery.

Conditional INSERT: conditional_insert_clause

Specify the conditional_insert_clause to perform a conditional multitable insert. The Oracle Server filters each insert_into_clause through the corresponding WHEN condition, which determines whether that insert_into_clause is executed. A single multitable insert statement can contain up to 127 WHEN clauses.

Conditional INSERT: ALL

If you specify ALL, the Oracle Server evaluates each WHEN clause regardless of the results of the evaluation of any other WHEN clause. For each WHEN clause whose condition evaluates to true, the Oracle Server executes the corresponding INTO clause list.

Multitable INSERT Statements (continued)

Conditional FIRST: INSERT

If you specify **FIRST**, the Oracle Server evaluates each **WHEN** clause in the order in which it appears in the statement. If the first **WHEN** clause evaluates to true, the Oracle Server executes the corresponding **INTO** clause and skips subsequent **WHEN** clauses for the given row.

Conditional INSERT: ELSE Clause

For a given row, if no **WHEN** clause evaluates to true:

- If you have specified an **ELSE** clause, the Oracle Server executes the **INTO** clause list associated with the **ELSE** clause.
- If you did not specify an **ELSE** clause, the Oracle Server takes no action for that row.

Restrictions on Multitable INSERT Statements

- You can perform multitable inserts only on tables, not on views or materialized views.
- You cannot perform a multitable insert into a remote table.
- You cannot specify a table collection expression when performing a multitable insert.
- In a multitable insert, all of the `insert_into_clauses` cannot combine to specify more than 999 target columns.

Unconditional INSERT ALL

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

ORACLE

6-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Unconditional INSERT ALL

The example in the slide inserts rows into both the `SAL_HISTORY` and the `MGR_HISTORY` tables. The `SELECT` statement retrieves the details of employee ID, hire date, salary, and manager ID of those employees whose employee ID is greater than 200 from the `EMPLOYEES` table. The details of the employee ID, hire date, and salary are inserted into the `SAL_HISTORY` table. The details of employee ID, manager ID and salary are inserted into the `MGR_HISTORY` table.

This `INSERT` statement is referred to as an unconditional `INSERT`, as no further restriction is applied to the rows that are retrieved by the `SELECT` statement. All the rows retrieved by the `SELECT` statement are inserted into the two tables, `SAL_HISTORY` and `MGR_HISTORY`. The `VALUES` clause in the `INSERT` statements specifies the columns from the `SELECT` statement that have to be inserted into each of the tables. Each row returned by the `SELECT` statement results in two insertions, one for the `SAL_HISTORY` table and one for the `MGR_HISTORY` table.

The feedback `8 rows created` can be interpreted to mean that a total of eight insertions were performed on the base tables `SAL_HISTORY` and `MGR_HISTORY`.

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.

ORACLE

6-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Conditional INSERT ALL

The problem statement for a conditional `INSERT ALL` statement is specified in the slide. The solution to the preceding problem is shown in the next page.

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.
```

ORACLE

6-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Conditional INSERT ALL (continued)

The example in the slide is similar to the example on the previous slide as it inserts rows into both the SAL_HISTORY and the MGR_HISTORY tables. The SELECT statement retrieves the details of employee ID, hire date, salary, and manager ID of those employees whose employee ID is greater than 200 from the EMPLOYEES table. The details of employee ID, hire date, and salary are inserted into the SAL_HISTORY table. The details of employee ID, manager ID, and salary are inserted into the MGR_HISTORY table.

This INSERT statement is referred to as a conditional ALL INSERT, as a further restriction is applied to the rows that are retrieved by the SELECT statement. From the rows that are retrieved by the SELECT statement, only those rows in which the value of the SAL column is more than 10000 are inserted in the SAL_HISTORY table, and similarly only those rows where the value of the MGR column is more than 200 are inserted in the MGR_HISTORY table.

Observe that unlike the previous example, where eight rows were inserted into the tables, in this example only four rows are inserted.

The feedback 4 rows created can be interpreted to mean that a total of four inserts were performed on the base tables, SAL_HISTORY and MGR_HISTORY.

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`.

ORACLE

6-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Conditional FIRST INSERT

The problem statement for a conditional `FIRST INSERT` statement is specified in the slide. The solution to the preceding problem is shown on the next page.

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.
```

ORACLE

6-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Conditional FIRST INSERT (continued)

The example in the slide inserts rows into more than one table, using one single INSERT statement. The SELECT statement retrieves the details of department ID, total salary, and maximum hire date for every department in the EMPLOYEES table.

This INSERT statement is referred to as a conditional FIRST INSERT, as an exception is made for the departments whose total salary is more than \$25,000. The condition WHEN SAL > 25000 is evaluated first. If the total salary for a department is more than \$25,000, then the record is inserted into the SPECIAL_SAL table irrespective of the hire date. If this first WHEN clause evaluates to true, the Oracle Server executes the corresponding INTO clause and skips subsequent WHEN clauses for this row.

For the rows that do not satisfy the first WHEN condition (WHEN SAL > 25000), the rest of the conditions are evaluated just as a conditional INSERT statement, and the records retrieved by the SELECT statement are inserted into the HIREDATE_HISTORY_00, or HIREDATE_HISTORY_99, or HIREDATE_HISTORY tables, based on the value in the HIREDATE column.

The feedback 8 rows created can be interpreted to mean that a total of eight INSERT statements were performed on the base tables SPECIAL_SAL, HIREDATE_HISTORY_00, HIREDATE_HISTORY_99, and HIREDATE_HISTORY.

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.

ORACLE

6-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Pivoting INSERT

Pivoting is an operation in which you need to build a transformation such that each record from any input stream, such as, a nonrelational database table, must be converted into multiple records for a more relational database table environment.

In order to solve the problem mentioned in the slide, you need to build a transformation such that each record from the original nonrelational database table, `SALES_SOURCE_DATA`, is converted into five records for the data warehouse's `SALES_INFO` table. This operation is commonly referred to as *pivoting*.

The problem statement for a pivoting INSERT statement is specified in the slide. The solution to the preceding problem is shown in the next page.

Pivoting INSERT

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

ORACLE

6-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Pivoting INSERT (continued)

In the example in the slide, the sales data is received from the nonrelational database table SALES_SOURCE_DATA, which is the details of the sales performed by a sales representative on each day of a week, for a week with a particular week ID.

```
DESC SALES_SOURCE_DATA
```

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
WEEK_ID		NUMBER(2)
SALES_MON		NUMBER(8,2)
SALES_TUE		NUMBER(8,2)
SALES_WED		NUMBER(8,2)
SALES_THUR		NUMBER(8,2)
SALES_FRI		NUMBER(8,2)

Pivoting INSERT (continued)

```
SELECT * FROM SALES_SOURCE_DATA;
```

EMPLOYEE_ID	WEEK_ID	SALES_MON	SALES_TUE	SALES_WED	SALES_THUR	SALES_FRI
176	6	2000	3000	4000	5000	6000

```
DESC SALES_INFO
```

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
WEEK		NUMBER(2)
SALES		NUMBER(8,2)

```
SELECT * FROM sales_info;
```

EMPLOYEE_ID	WEEK	SALES
176	6	2000
176	6	3000
176	6	4000
176	6	5000
176	6	6000

Observe in the preceding example that using a pivoting INSERT, one row from the SALES_SOURCE_DATA table is converted into five records for the relational table, SALES_INFO.

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.**

ORACLE

6-18

Copyright © Oracle Corporation, 2001. All rights reserved.

External Tables

An external table is a read-only table whose metadata is stored in the database but whose data is stored outside the database. Using the Oracle9i external table feature, you can use external data as a virtual table. This data can be queried and joined directly and in parallel without requiring the external data to be first loaded in the database. You can use SQL, PL/SQL, and Java to query the data in an external table.

The main difference between external tables and regular tables is that externally organized tables are read-only. No DML operations (`UPDATE`, `INSERT`, or `DELETE`) are possible, and no indexes can be created on them.

The means of defining the metadata for external tables is through the `CREATE TABLE . . . ORGANIZATION EXTERNAL` statement. This external table definition can be thought of as a view that is used for running any SQL query against external data without requiring that the external data first be loaded into the database.

The Oracle Server provides two major access drivers for external tables. One, the loader access driver, or `ORACLE_LOADER`, is used for reading of data from external files using the Oracle loader technology. This access driver allows the Oracle Server to access data from any data source whose format can be interpreted by the `SQL*Loader` utility. The other Oracle provided access driver, the import/export access driver, or `ORACLE_INTERNAL`, can be used for both the importing and exporting of data using a platform independent format.

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`

ORACLE

Creating an External Table

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

`TYPE access_driver_type` indicates the access driver of the external table. The access driver is the Application Programming Interface (API) that interprets the external data for the database. If you do not specify `TYPE`, Oracle uses the default access driver, `ORACLE_LOADER`.

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

`DEFAULT DIRECTORY` lets you specify one or more default directory objects corresponding to directories on the file system where the external data sources may reside. Default directories can also be used by the access driver to store auxiliary files such as error logs. Multiple default directories are permitted to facilitate load balancing on multiple disk drives.

The optional `ACCESS PARAMETERS` clause lets you assign values to the parameters of the specific access driver for this external table. Oracle does not interpret anything in this clause. It is up to the access driver to interpret this information in the context of the external data.

The `LOCATION` clause lets you specify one external locator for each external data source. Usually the `location_specifier` is a file, but it need not be. Oracle does not interpret this clause. It is up to the access driver to interpret this information in the context of the external data.

Example of Creating an External Table

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

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

ORACLE

6-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of Creating an External Table

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

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

Syntax

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

In the syntax:

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

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.
```

ORACLE

6-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of Creating an External Table (continued)

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

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

Records are delimited by new lines, and the fields are all terminated by a comma (,). The name of the file is: /flat_files/empl.txt

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

1. Create a directory object emp_dir as follows:

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

2. Run the CREATE TABLE command shown in the slide.

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

```
/flat_files/empl.txt
```

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

The PARALLEL clause enables five parallel execution servers to simultaneously scan the external data sources (files) when executing the INSERT INTO TABLE statement. For example, if PARALLEL=5 were specified, then more than one parallel execution server could be working on a data source. Because external tables can be very large, for performance reasons it is advisable to specify the PARALLEL clause, or a parallel hint for the query.

Example of Defining External Tables

The `REJECT LIMIT` clause specifies that if more than 200 conversion errors occur during a query of the external data, the query is aborted and an error returned. These conversion errors can arise when the access driver tries to transform the data in the data file to match the external table definition.

Once the `CREATE TABLE` command executes successfully, the external table `OLDEMP` can be described and queried like a relational table.

```
DESC oldemp
```

Name	Null?	Type
EMPNO		NUMBER
EMPNAME		CHAR(20)
BIRTHDATE		DATE

In the following example, the `INSERT INTO TABLE` statement generates a dataflow from the external data source to the Oracle SQL engine where data is processed. As data is extracted from the external table, it is transparently converted by the `ORACLE_ LOADER` access driver from its external representation into an equivalent Oracle native representation. The `INSERT` statement inserts data from the external table `OLDEMP` into the `BIRTHDAYS` table:

```
INSERT INTO birthdays(empno, empname, birthdate)
      SELECT      empno, empname, birthdate FROM oldemp;

2 rows created.
```

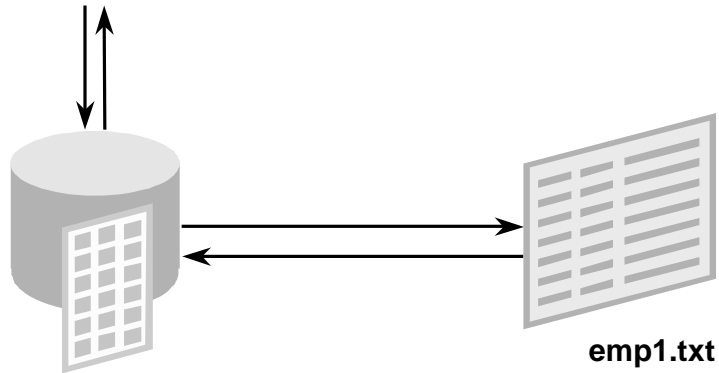
We can now select from the `BIRTHDAYS` table.

```
SELECT * FROM birthdays;
```

EMPNO	EMPNAME	BIRTHDATE
10	jones	11-DEC-34
20	smith	12-JUN-97

Querying External Tables

```
SELECT *  
FROM oldemp
```



ORACLE

6-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Querying External Tables

An external table does not describe any data that is stored in the database. Nor does it describe how data is stored in the external source. Instead, it describes how the external table layer needs to present the data to the server. It is the responsibility of the access driver and the external table layer to do the necessary transformations required on the data in the data file so that it matches the external table definition.

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

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

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

ORACLE

6-24

Copyright © Oracle Corporation, 2001. All rights reserved.

CREATE INDEX with CREATE TABLE Statement

In the example in the slide, the CREATE INDEX clause is used with the CREATE TABLE statement to create a primary key index explicitly. This is an enhancement provided with Oracle9i. You can now name your indexes at the time of PRIMARY key creation, unlike before where the Oracle Server would create an index, but you did not have any control over the name of the index. The following example illustrates this:

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

Table created.

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

INDEX_NAME	TABLE_NAME
SYS_C002835	EMP_UNNAMED_INDEX

Observe that the Oracle Server gives a name to the Index that it creates for the PRIMARY KEY column. But this name is cryptic and not easily understood. With Oracle9i, you can name your PRIMARY KEY column indexes, as you create the table with the CREATE TABLE statement. However, prior to Oracle9i, if you named your primary key constraint at the time of constraint creation, the index would also be created with the same name as the constraint name.

Summary

In this lesson, you should have learned how to:

- Use the `INSERT...SELECT` statement to insert rows into multiple tables as part of a single DML statement
- Create external tables
- Name indexes using the `CREATE INDEX` statement along with the `CREATE TABLE` statement

ORACLE

6-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

Oracle9i introduces the following types of multitable `INSERT` statements.

- Unconditional `INSERT`
- Conditional `ALL INSERT`
- Conditional `FIRST INSERT`
- Pivoting `INSERT`

Use the `external_table_clause` to create an external table, which is a read-only table whose metadata is stored in the database but whose data is stored outside the database. You can use external tables to query data without first loading it into the database.

With Oracle9i, you can name your `PRIMARY KEY` column indexes as you create the table with the `CREATE TABLE` statement.

Practice 6 Overview

This practice covers the following topics:

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

ORACLE

6-26

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 6 Overview

In this practice, you write multitable inserts and use the `CREATE INDEX` command at the time of table creation, along with the `CREATE TABLE` command.

Practice 6

1. Run the `cre_sal_history.sql` script in the lab folder to create the `SAL_HISTORY` table.
2. Display the structure of the `SAL_HISTORY` table.

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
HIRE_DATE		DATE
SALARY		NUMBER(8,2)

3. Run the `cre_mgr_history.sql` script in the lab folder to create the `MGR_HISTORY` table.
4. Display the structure of the `MGR_HISTORY` table.

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
MANAGER_ID		NUMBER(6)
SALARY		NUMBER(8,2)

5. Run the `cre_special_sal.sql` script in the lab folder to create the `SPECIAL_SAL` table.
6. Display the structure of the `SPECIAL_SAL` table.

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
SALARY		NUMBER(8,2)

7. a. Write a query to do the following:
 - Retrieve the details of the employee ID, hire date, salary, and manager ID of those employees whose employee ID is less than 125 from the `EMPLOYEES` table.
 - If the salary is more than \$20,000, insert the details of employee ID and salary into the `SPECIAL_SAL` table.
 - Insert the details of employee ID, hire date, salary into the `SAL_HISTORY` table.
 - Insert the details of the employee ID, manager ID, and salary into the `MGR_HISTORY` table.

Practice 6 (continued)

b. Display the records from the SPECIAL_SAL table.

EMPLOYEE_ID	SALARY
100	24000

c. Display the records from the SAL_HISTORY table.

EMPLOYEE_ID	HIRE_DATE	SALARY
101	21-SEP-89	17000
102	13-JAN-93	17000
103	03-JAN-90	9000
104	21-MAY-91	6000
107	07-FEB-99	4200
124	16-NOV-99	5800

6 rows selected.

d. Display the records from the MGR_HISTORY table.

EMPLOYEE_ID	MANAGER_ID	SALARY
101	100	17000
102	100	17000
103	102	9000
104	103	6000
107	103	4200
124	100	5800

6 rows selected.

Practice 6 (continued)

- 8.a. Run the `cre_sales_source_data.sql` script in the lab folder to create the `SALES_SOURCE_DATA` table.
- b. Run the `ins_sales_source_data.sql` script in the lab folder to insert records into the `SALES_SOURCE_DATA` table.
- c. Display the structure of the `SALES_SOURCE_DATA` table.

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
WEEK_ID		NUMBER(2)
SALES_MON		NUMBER(8,2)
SALES_TUE		NUMBER(8,2)
SALES_WED		NUMBER(8,2)
SALES_THUR		NUMBER(8,2)
SALES_FRI		NUMBER(8,2)

- d. Display the records from the `SALES_SOURCE_DATA` table.

EMPLOYEE_ID	WEEK_ID	SALES_MON	SALES_TUE	SALES_WED	SALES_THUR	SALES_FRI
178	6	1750	2200	1500	1500	3000

- e. Run the `cre_sales_info.sql` script in the lab folder to create the `SALES_INFO` table.
- f. Display the structure of the `SALES_INFO` table.

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
WEEK		NUMBER(2)
SALES		NUMBER(8,2)

Practice 6 (continued)

g. Write a query to do the following:

Retrieve the details of employee ID, week ID, sales on Monday, sales on Tuesday, sales on Wednesday, sales on Thursday, and sales on Friday from the SALES_SOURCE_DATA table.

Build a transformation such that each record retrieved from the SALES_SOURCE_DATA table is converted into multiple records for the SALES_INFO table.

Hint: Use a pivoting INSERT statement.

h. Display the records from the SALES_INFO table.

EMPLOYEE_ID	WEEK	SALES
178	6	1750
178	6	2200
178	6	1500
178	6	1500
178	6	3000

9. a. Create the DEPT_NAMED_INDEX table based on the following table instance chart. Name the index for the PRIMARY KEY column as DEPT_PK_IDX.

COLUMN Name	Deptno	Dname
Primary Key	Yes	
Datatype	Number	VARCHAR2
Length	4	30

b. Query the USER_INDEXES table to display the INDEX_NAME for the DEPT_NAMED_INDEX table.

INDEX_NAME	TABLE_NAME
DEPT_PK_IDX	DEPT_NAMED_INDEX

A

Practice Solutions

Practice 1 Solutions

1. List the department IDs for departments that do not contain the job ID ST_CLERK, using SET operators.

```
SELECT department_id
FROM departments
MINUS
SELECT department_id
FROM employees
WHERE job_id = 'ST_CLERK';
```

2. Display the country ID and the name of the countries that have no departments located in them, using SET operators.

```
SELECT country_id, country_name
FROM countries
MINUS
SELECT l.country_id, c.country_name
FROM locations l, countries c
WHERE l.country_id = c.country_id;
```

3. Produce a list of jobs for departments 10, 50, and 20, in that order. Display job ID and department ID, using SET operators.

```
COLUMN dummy NOPRINT
SELECT job_id, department_id, 'x' dummy
FROM employees
WHERE department_id = 10
UNION
SELECT job_id, department_id, 'y'
FROM employees
WHERE department_id = 50
UNION
SELECT job_id, department_id, 'z'
FROM employees
WHERE department_id = 20
ORDER BY 3;
COLUMN dummy PRINT
```


Practice 1 Solutions (continued)

4. List the employee IDs and job IDs of those employees who currently have the job title that they held before beginning their tenure with the company.

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

5. Write a compound query that lists the following:

- Last names and department ID of all the employees from the EMPLOYEES table, regardless of whether or not they belong to any department
- Department ID and department name of all the departments from the DEPARTMENTS table, regardless of whether or not they have employees working in them

```
SELECT last_name,department_id,TO_CHAR(null)
FROM    employees
UNION
SELECT TO_CHAR(null),department_id,department_name
FROM    departments;
```

Practice 2 Solutions

1. Alter the session to set the NLS_DATE_FORMAT to DD-MON-YYYY HH24:MI:SS.

```
ALTER SESSION SET NLS_DATE_FORMAT =  
'DD-MON-YYYY HH24:MI:SS';
```

2. a. Write queries to display the time zone offsets (TZ_OFFSET) for the following time zones.

US/Pacific-New

```
SELECT TZ_OFFSET ('US/Pacific-New') from dual;
```

Singapore

```
SELECT TZ_OFFSET ('Singapore') from dual;
```

Egypt

```
SELECT TZ_OFFSET ('Egypt') from dual;
```

- b. Alter the session to set the TIME_ZONE parameter value to the time zone offset of US/Pacific-New.

```
ALTER SESSION SET TIME_ZONE = '-7:00';
```

- c. Display the CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP for this session.

Note: The output might be different based on the date when the command is executed.

```
SELECT CURRENT_DATE, CURRENT_TIMESTAMP, LOCALTIMESTAMP  
FROM DUAL;
```

- d. Alter the session to set the TIME_ZONE parameter value to the time zone offset of Singapore.

```
ALTER SESSION SET TIME_ZONE = '+8:00';
```

- e. Display the CURRENT_DATE, CURRENT_TIMESTAMP, LOCALTIMESTAMP for this session.

Note: The output might be different, based on the date when the command is executed.

```
SELECT CURRENT_DATE, CURRENT_TIMESTAMP, LOCALTIMESTAMP  
FROM DUAL;
```

3. Write a query to display the DBTIMEZONE and SESSIONTIMEZONE.

```
SELECT DBTIMEZONE,SESSIONTIMEZONE  
FROM DUAL;
```

Practice 2 Solutions (continued)

4. Write a query to extract the YEAR from HIRE_DATE column of the EMPLOYEES table for those employees who work in department 80.

```
SELECT last_name, EXTRACT (YEAR FROM HIRE_DATE)
FROM employees
WHERE department_id = 80;
```

5. Alter the session to set the NLS_DATE_FORMAT to DD-MON-YYYY.

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY';
```

Practice 3 Solutions

1. Write a query to display the following for those employees whose manager ID is less than 120:
 - Manager ID
 - Job ID and total salary for every job ID for employees who report to the same manager
 - Total salary of those managers
 - Total salary of those managers, irrespective of the job IDs

```
SELECT manager_id, job_id, sum(salary)
FROM   employees
WHERE  manager_id < 120
GROUP BY ROLLUP(manager_id, job_id);
```

2. Observe the output from question 1. Write a query using the GROUPING function to determine whether the NULL values in the columns corresponding to the GROUP BY expressions are caused by the ROLLUP operation.

```
SELECT manager_id MGR , job_id JOB,
sum(salary), GROUPING(manager_id), GROUPING(job_id)
FROM   employees
WHERE  manager_id < 120
GROUP BY ROLLUP(manager_id, job_id);
```

3. Write a query to display the following for those employees whose manager ID is less than 120 :

- Manager ID
- Job and total salaries for every job for employees who report to the same manager
- Total salary of those managers
- Cross-tabulation values to display the total salary for every job, irrespective of the manager
- Total salary irrespective of all job titles

```
SELECT manager_id, job_id, sum(salary)
FROM   employees
WHERE  manager_id < 120
GROUP BY CUBE(manager_id, job_id);
```

Practice 3 Solutions (continued)

4. Observe the output from question 3. Write a query using the GROUPING function to determine whether the NULL values in the columns corresponding to the GROUP BY expressions are caused by the CUBE operation.

```
SELECT manager_id MGR ,job_id JOB,
       sum(salary),GROUPING(manager_id),GROUPING(job_id)
FROM   employees
WHERE  manager_id < 120
GROUP BY CUBE(manager_id,job_id);
```

5. Using GROUPING SETS, write a query to display the following groupings :
- department_id, manager_id, job_id
 - department_id, job_id
 - Manager_id, job_id

The query should calculate the sum of the salaries for each of these groups.

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

Practice 4 Solutions

1. Write a query to display the last name, department number, and salary of any employee whose department number and salary both match the department number and salary of any employee who earns a commission.

```
SELECT last_name, department_id, salary
FROM   employees
WHERE  (salary, department_id) IN
      (SELECT salary, department_id
       FROM   employees
       WHERE  commission_pct IS NOT NULL);
```

2. Display the last name, department name, and salary of any employee whose salary and commission match the salary and commission of any employee located in location ID1700.

```
SELECT e.last_name, d.department_name, e.salary
FROM   employees e, departments d
WHERE  e.department_id = d.department_id
AND    (salary, NVL(commission_pct,0)) IN
      (SELECT salary, NVL(commission_pct,0)
       FROM   employees e, departments d
       WHERE  e.department_id = d.department_id
       AND    d.location_id = 1700);
```

3. Create a query to display the last name, hire date, and salary for all employees who have the same salary and commission as Kochhar.

Note: Do not display Kochhar in the result set.

```
SELECT last_name, hire_date, salary
FROM   employees
WHERE  (salary, NVL(commission_pct,0)) IN
      (SELECT salary, NVL(commission_pct,0)
       FROM   employees
       WHERE  last_name = 'Kochhar')
AND last_name != 'Kochhar';
```

4. Create a query to display the employees who earn a salary that is higher than the salary of all of the sales managers (JOB_ID = 'SA_MAN'). Sort the results on salary from highest to lowest.

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  salary > ALL
      (SELECT salary
       FROM   employees
       WHERE  job_id = 'SA_MAN')
ORDER BY salary DESC;
```

Practice 4 Solutions (continued)

5. Display the details of the employee ID, last name, and department ID of those employees who live in cities whose name begins with *T*.

```
SELECT employee_id, last_name, department_id
FROM   employees
WHERE  department_id IN (SELECT department_id
                        FROM departments
                        WHERE location_id IN
                              (SELECT location_id
                               FROM locations
                               WHERE city LIKE 'T%'));
```

6. Write a query to find all employees who earn more than the average salary in their departments. Display last name, salary, department ID, and the average salary for the department. Sort by average salary. Use aliases for the columns retrieved by the query as shown in the sample output.

```
SELECT e.last_name ename, e.salary salary,
       e.department_id deptno, AVG(a.salary) dept_avg
FROM   employees e, employees a
WHERE  e.department_id = a.department_id
AND    e.salary > (SELECT AVG(salary)
                  FROM   employees
                  WHERE  department_id = e.department_id )
GROUP BY e.last_name, e.salary, e.department_id
ORDER BY AVG(a.salary);
```

7. Find all employees who are not supervisors.
a. First do this by using the NOT EXISTS operator.

```
SELECT outer.last_name
FROM   employees outer
WHERE  NOT EXISTS (SELECT 'X'
                  FROM employees inner
                  WHERE inner.manager_id =
                        outer.employee_id);
```

Practice 4 Solutions (continued)

- b. Can this be done by using the NOT IN operator? How, or why not?

```
SELECT outer.last_name
FROM   employees outer
WHERE  outer.employee_id
NOT IN (SELECT inner.manager_id
        FROM   employees inner);
```

This alternative solution is not a good one. The subquery picks up a NULL value, so the entire query returns no rows. The reason is that all conditions that compare a NULL value result in NULL. Whenever NULL values are likely to be part of the value set, *do not* use NOT IN as a substitute for NOT EXISTS.

8. Write a query to display the last names of the employees who earn less than the average salary in their departments.

```
SELECT last_name
FROM   employees outer
WHERE  outer.salary < (SELECT AVG(inner.salary)
                      FROM employees inner
                      WHERE inner.department_id
                        = outer.department_id);
```

9. Write a query to display the last names of employees who have one or more coworkers in their departments with later hire dates but higher salaries.

```
SELECT last_name
FROM   employees outer
WHERE  EXISTS (SELECT 'X'
              FROM employees inner
              WHERE inner.department_id =
                    outer.department_id
              AND inner.hire_date > outer.hire_date
              AND inner.salary > outer.salary);
```

10. Write a query to display the employee ID, last names, and department names of all employees.

Note: Use a scalar subquery to retrieve the department name in the SELECT statement.

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


Practice 4 Solutions (continued)

11. Write a query to display the department names of those departments whose total salary cost is above one-eighth (1/8) of the total salary cost of the whole company. Use the WITH clause to write this query. Name the query SUMMARY.

```
WITH
summary AS (
    SELECT d.department_name, SUM(e.salary) AS dept_total
    FROM employees e, departments d
    WHERE e.department_id = d.department_id
    GROUP BY d.department_name)
SELECT department_name, dept_total
FROM summary
WHERE dept_total > (
    SELECT SUM(dept_total) * 1/8
    FROM summary )
ORDER BY dept_total DESC;
```

Practice 5 Solutions

1. Look at the following outputs. Are these outputs the result of a hierarchical query? Explain why or why not.

Exhibit 1: This is not a hierarchical query; the report simply has a descending sort on SALARY.

Exhibit 2: This is not a hierarchical query; there are two tables involved.

Exhibit 3: Yes, this is most definitely a hierarchical query as it displays the tree structure representing the management reporting line from the EMPLOYEES table.

2. Produce a report showing an organization chart for Mourgos's department. Print last names, salaries, and department IDs.

```
SELECT last_name, salary, department_id
FROM employees
START WITH last_name = 'Mourgos'
CONNECT BY PRIOR employee_id = manager_id;
```

3. Create a report that shows the hierarchy of the managers for the employee Lorentz. Display his immediate manager first.

```
SELECT last_name
FROM employees
WHERE last_name != 'Lorentz'
START WITH last_name = 'Lorentz'
CONNECT BY PRIOR manager_id = employee_id;
```

4. Create an indented report showing the management hierarchy starting from the employee whose LAST_NAME is Kochhar. Print the employee's last name, manager ID, and department ID. Give alias names to the columns as shown in the sample output.

```
COLUMN name FORMAT A20
SELECT LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-2, '_')
       name, manager_id mgr, department_id deptno
FROM employees
START WITH last_name = 'Kochhar'
CONNECT BY PRIOR employee_id = manager_id
/
COLUMN name CLEAR
```

Practice 5 Solutions (continued)

If you have time, complete the following exercises:

5. Produce a company organization chart that shows the management hierarchy. Start with the person at the top level, exclude all people with a job ID of IT_PROG, and exclude De Haan and those employees who report to De Haan.

```
SELECT last_name, employee_id, manager_id
FROM   employees
WHERE  job_id != 'IT_PROG'
START WITH manager_id IS NULL
CONNECT BY PRIOR employee_id = manager_id
AND last_name != 'De Haan';
```

Practice 6 Solutions

1. Run the `cre_sal_history.sql` script in the lab folder to create the `SAL_HISTORY` table.

```
@ \lab\cre_sal_history.sql
```

2. Display the structure of the `SAL_HISTORY` table.

```
DESC sal_history
```

3. Run the `cre_mgr_history.sql` script in the lab folder to create the `MGR_HISTORY` table.

```
@ \lab\cre_mgr_history.sql
```

4. Display the structure of the `MGR_HISTORY` table.

```
DESC mgr_history
```

5. Run the `cre_special_sal.sql` script in the lab folder to create the `SPECIAL_SAL` table.

```
@ \lab\cre_special_sal.sql
```

6. Display the structure of the `SPECIAL_SAL` table.

```
DESC special_sal
```

7. a. Write a query to do the following:

- Retrieve the details of the employee ID, hire date, salary, and manager ID of those employees whose employee ID is less than 125 from the `EMPLOYEES` table.
- If the salary is more than \$20,000, insert the details of employee ID and salary into the `SPECIAL_SAL` table.
- Insert the details of the employee ID, hire date, and salary into the `SAL_HISTORY` table.
- Insert the details of the employee ID, manager ID, and salary into the `MGR_HISTORY` table.

```
INSERT ALL
WHEN SAL > 20000 THEN
  INTO special_sal VALUES (EMPID, SAL)
ELSE
  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 < 125;
```

Practice 6 Solutions (continued)

- b. Display the records from the SPECIAL_SAL table.

```
SELECT * FROM special_sal;
```

- c. Display the records from the SAL_HISTORY table.

```
SELECT * FROM sal_history;
```

- d. Display the records from the MGR_HISTORY table.

```
SELECT * FROM mgr_history;
```

8. a. Run the cre_sales_source_data.sql script in the lab folder to create the SALES_SOURCE_DATA table.

```
@ \lab\cre_sales_source_data.sql
```

- b. Run the ins_sales_source_data.sql script in the lab folder to insert records into the SALES_SOURCE_DATA table.

```
@ \lab\ins_sales_source_data.sql
```

- c. Display the structure of the SALES_SOURCE_DATA table.

```
DESC sales_source_data
```

- d. Display the records from the SALES_SOURCE_DATA table.

```
SELECT * FROM SALES_SOURCE_DATA;
```

- e. Run the cre_sales_info.sql script in the lab folder to create the SALES_INFO table.

```
@ \lab\cre_sales_info.sql
```

- f. Display the structure of the SALES_INFO table.

```
DESC sales_info
```

- g. Write a query to do the following:

- Retrieve the details of the employee ID, week ID, sales on Monday, sales on Tuesday, sales on Wednesday, sales on Thursday, and sales on Friday from the SALES_SOURCE_DATA table.
- Build a transformation such that each record retrieved from the SALES_SOURCE_DATA table is converted into multiple records for the SALES_INFO table.

Hint: Use a pivoting INSERT statement.

Practice 6 Solutions (continued)

```
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;
```

h. Display the records from the SALES_INFO table.

```
SELECT * FROM sales_info;
```

9. a. Create the DEPT_NAMED_INDEX table based on the following table instance chart. Name the index for the PRIMARY KEY column as DEPT_PK_IDX.

COLUMN Name	Deptno	Dname
Primary Key	Yes	
Data type	Number	VARCHAR2
Length	4	30

```
CREATE TABLE DEPT_NAMED_INDEX
(deptno NUMBER(4)
PRIMARY KEY USING INDEX
(CREATE INDEX dept_pk_idx ON
DEPT_NAMED_INDEX(deptno)),
dname VARCHAR2(30));
```

- b. Query the USER_INDEXES table to display the INDEX_NAME for the DEPT_NAMED_INDEX table.

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

Practice C Solutions

1. Write a script to describe and select the data from your tables. Use `CHR(10)` in the select list with the concatenation operator (`||`) to generate a line feed in your report. Save the output of the script into `my_file1.sql`. To save the file, select the `FILE` option for the output, and execute the code. Remember to save the file with a `.sql` extension. To execute the `my_file1.sql`, browse to locate the script, load the script, and execute the script.

```
SET PAGESIZE 0

SELECT 'DESC ' || table_name || CHR(10) ||
       'SELECT * FROM ' || table_name || ';'
FROM   user_tables
/
SET PAGESIZE 24
SET LINESIZE 100
```

2. Use SQL to generate SQL statements that revoke user privileges. Use the data dictionary views `USER_TAB_PRIVS_MADE` and `USER_COL_PRIVS_MADE`.
 - a. Execute the script `\lab\privs.sql` to grant privileges to the user `SYSTEM`.
 - b. Query the data dictionary views to check the privileges. In the sample output shown, note that the data in the `GRANTOR` column can vary depending on who the `GRANTOR` is. Also the last column that has been truncated is the `GRANTABLE` column.

```
COLUMN      grantee  FORMAT  A10
COLUMN      table_name  FORMAT  A10
COLUMN      column_name  FORMAT  A10
COLUMN      grantor   FORMAT  A10
COLUMN      privilege   FORMAT  A10
SELECT *
FROM      user_tab_privs_made
WHERE     grantee = 'SYSTEM';

SELECT *
FROM      user_col_privs_made
WHERE     grantee = 'SYSTEM';
```

Practice C Solutions (continued)

- c. Produce a script to revoke the privileges. Save the output of the script into `my_file2.sql`. To save the file, select the `FILE` option for the output, and execute the code. Remember to save the file with a `.sql` extension. To execute the `my_file2.sql`, browse to locate the script, load the script, and execute the script.

```
SET VERIFY OFF
SET PAGESIZE 0
```

```
SELECT      'REVOKE ' || privilege || ' ON ' ||
table_name || ' FROM system;'
FROM    user_tab_privs_made
WHERE    grantee = 'SYSTEM'
/
SELECT      DISTINCT      'REVOKE ' || privilege || ' ON ' ||
table_name || ' FROM system;'
FROM    user_col_privs_made
WHERE    grantee = 'SYSTEM'
/
```

```
SET VERIFY ON
SET PAGESIZE 24
```

B

Table Descriptions and Data

COUNTRIES Table

DESCRIBE countries

Name	Null?	Type
COUNTRY_ID	NOT NULL	CHAR(2)
COUNTRY_NAME		VARCHAR2(40)
REGION_ID		NUMBER

SELECT * FROM countries;

CO	COUNTRY_NAME	REGION_ID
CA	Canada	2
DE	Germany	1
UK	United Kingdom	1
US	United States of America	2

DEPARTMENTS Table

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)

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.

EMPLOYEES Table

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)

SELECT * FROM employees;

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

20 rows selected.

EMPLOYEES Table (continued)

JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
AD_PRES	24000			90
AD_VP	17000		100	90
AD_VP	17000		100	90
IT_PROG	9000		102	60
IT_PROG	6000		103	60
IT_PROG	4200		103	60
ST_MAN	5800		100	50
ST_CLERK	3500		124	50
ST_CLERK	3100		124	50
ST_CLERK	2600		124	50
ST_CLERK	2500		124	50
SA_MAN	10500	.2	100	80
SA_REP	11000	.3	149	80
SA_REP	8600	.2	149	80
SA_REP	7000	.15	149	
AD_ASST	4400		101	10
MK_MAN	13000		100	20
MK_REP	6000		201	20
AC_MGR	12000		101	110
AC_ACCOUNT	8300		205	110

20 rows selected.

JOBS Table

DESCRIBE jobs

Name	Null?	Type
JOB_ID	NOT NULL	VARCHAR2(10)
JOB_TITLE	NOT NULL	VARCHAR2(35)
MIN_SALARY		NUMBER(6)
MAX_SALARY		NUMBER(6)

SELECT * FROM jobs;

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_PRES	President	20000	40000
AD_VP	Administration Vice President	15000	30000
AD_ASST	Administration Assistant	3000	6000
AC_MGR	Accounting Manager	8200	16000
AC_ACCOUNT	Public Accountant	4200	9000
SA_MAN	Sales Manager	10000	20000
SA_REP	Sales Representative	6000	12000
ST_MAN	Stock Manager	5500	8500
ST_CLERK	Stock Clerk	2000	5000
IT_PROG	Programmer	4000	10000
MK_MAN	Marketing Manager	9000	15000
MK_REP	Marketing Representative	4000	9000

12 rows selected.

JOB_GRADES Table

DESCRIBE job_grades

Name	Null?	Type
GRADE_LEVEL		VARCHAR2(3)
LOWEST_SAL		NUMBER
HIGHEST_SAL		NUMBER

SELECT * FROM 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.

JOB_HISTORY Table

```
DESCRIBE job_history
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
START_DATE	NOT NULL	DATE
END_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
DEPARTMENT_ID		NUMBER(4)

```
SELECT * FROM job_history;
```

EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID
102	13-JAN-93	24-JUL-98	IT_PROG	60
101	21-SEP-89	27-OCT-93	AC_ACCOUNT	110
101	28-OCT-93	15-MAR-97	AC_MGR	110
201	17-FEB-96	19-DEC-99	MK_REP	20
114	24-MAR-98	31-DEC-99	ST_CLERK	50
122	01-JAN-99	31-DEC-99	ST_CLERK	50
200	17-SEP-87	17-JUN-93	AD_ASST	90
176	24-MAR-98	31-DEC-98	SA_REP	80
176	01-JAN-99	31-DEC-99	SA_MAN	80
200	01-JUL-94	31-DEC-98	AC_ACCOUNT	90

10 rows selected.

LOCATIONS Table

DESCRIBE locations

Name	Null?	Type
LOCATION_ID	NOT NULL	NUMBER(4)
STREET_ADDRESS		VARCHAR2(40)
POSTAL_CODE		VARCHAR2(12)
CITY	NOT NULL	VARCHAR2(30)
STATE_PROVINCE		VARCHAR2(25)
COUNTRY_ID		CHAR(2)

SELECT * FROM locations;

LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY	STATE_PROVINCE	CO
1400	2014 Jabberwocky Rd	26192	Southlake	Texas	US
1500	2011 Interiors Blvd	99236	South San Francisco	California	US
1700	2004 Charade Rd	98199	Seattle	Washington	US
1800	460 Bloor St. W.	ON M5S 1X8	Toronto	Ontario	CA
2500	Magdalen Centre, The Oxford Science Park	OX9 9ZB	Oxford	Oxford	UK

REGIONS Table

DESCRIBE regions

Name	Null?	Type
REGION_ID	NOT NULL	NUMBER
REGION_NAME		VARCHAR2(25)

SELECT * FROM regions;

REGION_ID	REGION_NAME
1	Europe
2	Americas
3	Asia
4	Middle East and Africa



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**

ORACLE

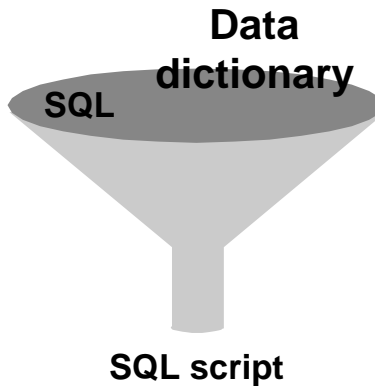
C-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this appendix, you learn how to write a SQL script to generates a SQL script.

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**

ORACLE

C-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Using SQL to Generate SQL

SQL can be a powerful tool to generate other SQL statements. In most cases this involves writing a script file. You can use SQL from SQL to:

- Avoid repetitive coding
- Access information from the data dictionary
- Drop or re-create database objects
- Generate dynamic predicates that contain run-time parameters

The examples used in this lesson involve selecting information from the data dictionary. The data dictionary is a collection of tables and views that contain information about the database. This collection is created and maintained by the Oracle Server. All data dictionary tables are owned by the SYS user. Information stored in the data dictionary includes names of the Oracle Server users, privileges granted to users, database object names, table constraints, and audit information. There are four categories of data dictionary views. Each category has a distinct prefix that reflects its intended use.

Prefix	Description
USER_	Contains details of objects owned by the user
ALL_	Contains details of objects to which the user has been granted access rights, in addition to objects owned by the user
DBA_	Contains details of users with DBA privileges to access any object in the database
V\$_	Stored information about database server performance and locking; available only to the DBA

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

C-4

Copyright © Oracle Corporation, 2001. All rights reserved.

A Basic Script

The example in the slide produces a report with `CREATE TABLE` statements from every table you own. Each `CREATE TABLE` statement produced in the report includes the syntax to create a table using the table name with a suffix of `_test` and having only the structure of the corresponding existing table. The old table name is obtained from the `TABLE_NAME` column of the data dictionary view `USER_TABLES`.

The next step is to enhance the report to automate the process.

Note: You can query the data dictionary tables to view various database objects that you own. The data dictionary views frequently used include:

- `USER_TABLES`: Displays description of the user's own tables
- `USER_OBJECTS`: Displays all the objects owned by the user
- `USER_TAB_PRIVS_MADE`: Displays all grants on objects owned by the user
- `USER_COL_PRIVS_MADE`: Displays all grants on columns of objects owned by the user

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.

ORACLE

C-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Controlling the Environment

In order to execute the SQL statements that are generated, you must capture them in a spool file that can then be run. You must also plan to clean up the output that is generated and make sure that you suppress elements such as headings, feedback messages, top titles, and so on. You can accomplish all of this by using *iSQL*Plus* commands.

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
```

ORACLE

C-6

Copyright © Oracle Corporation, 2001. All rights reserved.

The Complete Picture

The output of the command on the slide is saved into a file called `dropem.sql` using the File Output option in *iSQL*Plus*. This file contains the following data. This file can now be started from the *iSQL*Plus* by locating the script file, loading it, and executing it.

'DROPTABLE' OBJECT_NAME ';'
DROP TABLE COUNTRIES;
DROP TABLE DEPARTMENTS;
DROP TABLE EMPLOYEES;
DROP TABLE JOBS;
DROP TABLE JOB_HISTORY;
DROP TABLE LOCATIONS;
DROP TABLE REGIONS;

Note: By default, files are spooled into the `ORACLE_HOME\ORANT\BIN` folder in Windows NT.

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
```

ORACLE

C-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Dumping Table Contents to a File

Sometimes it is useful to have the values for the rows of a table in a text file in the format of an `INSERT INTO VALUES` statement. This script can be run to populate the table, in case the table has been dropped accidentally.

The example in the slide produces `INSERT` statements for the `DEPARTMENTS_TEST` table, captured in the `data.sql` file using the `File Output` option in *iSQL*Plus*.

The contents of the `data.sql` script file are as follows:

```
INSERT INTO departments_test VALUES
  (10, 'Administration', 1700);
INSERT INTO departments_test VALUES
  (20, 'Marketing', 1800);
INSERT INTO departments_test VALUES
  (50, 'Shipping', 1500);
INSERT INTO departments_test VALUES
  (60, 'IT', 1400);
...
```

Dumping the Contents of a Table to a File

Source	Result
<code>'''X'''</code>	<code>'X'</code>
<code>'''</code>	<code>'</code>
<code>''' department_name '''</code>	<code>'Administration'</code>
<code>''' , '''</code>	<code>' , '</code>
<code>'''); '</code>	<code>'); '</code>

ORACLE

C-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Dumping Table Contents to a File (continued)

You may have noticed the large number of single quotes in the slide on the previous page. A set of four single quotes produces one single quote in the final statement. Also remember that character and date values must be surrounded by quotes.

Within a string, to display one single quote, you need to prefix it with another single quote. For example, in the fifth example in the slide, the surrounding quotes are for the entire string. The second quote acts as a prefix to display the third quote. Thus the result is one single quote followed by the parenthesis followed by the semicolon.

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;
```

;

ORACLE

C-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Generating a Dynamic Predicate

The example in the slide generates a `SELECT` statement that retrieves data of all employees in a department who were hired on a specific day. The script generates the `WHERE` clause dynamically.

Note: Once the user variable is in place, you need to use the `UNDEFINE` command to delete it.

The first `SELECT` statement prompts you to enter the department number. If you do not enter any department number, the department number is treated as null by the `DECODE` function, and the user is then prompted for the hire date. If you do not enter any hire date, the hire date is treated as null by the `DECODE` function and the dynamic `WHERE` clause that is generated is also a null, which causes the second `SELECT` statement to retrieve all rows from the `EMPLOYEES` table.

Note: The `NEW_V[ALUE]` variable specifies a variable to hold a column value. You can reference the variable in `TTITLE` commands. Use `NEW_VALUE` to display column values or the date in the top title. You must include the column in a `BREAK` command with the `SKIP PAGE` action. The variable name cannot contain a pound sign (`#`). `NEW_VALUE` is useful for master/detail reports in which there is a new master record for each page.

Generating a Dynamic Predicate (continued)

Note: Here, the hire date must be entered in DD-MON-YYYY format.

The SELECT statement in the previous slide can be interpreted as follows:

```
IF (<<deptno>> is not entered) THEN
  IF (<<hiredate>> is not entered) THEN
    return empty string
  ELSE
    return the string 'WHERE hire_date = TO_DATE('<<hiredate>>', 'DD-MON-YYYY')'
  ELSE
    IF (<<hiredate>> is not entered) THEN
      return the string 'WHERE department_id = <<deptno>> entered'
    ELSE
      return the string 'WHERE department_id = <<deptno>> entered
        AND hire_date = TO_DATE(' <<hiredate>>', 'DD-MON-YYYY')'
    END IF
  END IF
```

The returned string becomes the value of the variable DYN_WHERE_CLAUSE, that will be used in the second SELECT statement.

When the first example on the slide is executed, the user is prompted for the values for DEPTNO and

ORACLE

iSQL*Plus



Define Substitution Variables

"deptno"
"hiredate"

Submit for Execution

Cancel

MY_COL
WHERE department_id = 10AND hire_date = TO_DATE('17-SEP-1987','DD-MON-YYYY')

LAST_NAME
Whalen

When the second example on the slide is executed, the following output is generated:

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.**

ORACLE

C-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

SQL can be used to generate SQL scripts. These scripts can be used to avoid repetitive coding, drop or re-create objects, get help from the data dictionary, and generate dynamic predicates that contain run-time parameters.

*iSQL*Plus* commands can be used to capture the reports generated by the SQL statements and clean up the output that is generated, such as suppressing headings, feedback messages, and so on.

Practice C 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[®]

C-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice C Overview

In this practice, you gain practical experience in writing SQL to generate SQL.

Practice C

1. Write a script to describe and select the data from your tables. Use `CHR(10)` in the select list with the concatenation operator (`||`) to generate a line feed in your report. Save the output of the script into `my_file1.sql`. To save the file, select `File` option for the output and execute the code. Remember to save the file with a `.sql` extension. To execute the `my_file1.sql`, browse to locate the script, load the script, and execute the script.
2. Use SQL to generate SQL statements that revoke user privileges. Use the data dictionary views `USER_TAB_PRIVS_MADE` and `USER_COL_PRIVS_MADE`.
 - a. Execute the script `\Lab\privs.sql` to grant privileges to the user `SYSTEM`.
 - b. Query the data dictionary views to check the privileges. In the sample output shown, note that the data in the `GRANTOR` column can vary depending on who the `GRANTOR` is. Also the last column that has been truncated is the `GRANTABLE` column.

GRANTEE	TABLE_NAME	GRANTOR	PRIVILEGE	GRA	HIE
SYSTEM	DEPARTMENT S	SQL2	ALTER	NO	NO
SYSTEM	DEPARTMENT S	SQL2	DELETE	NO	NO
SYSTEM	DEPARTMENT S	SQL2	INDEX	NO	NO
SYSTEM	DEPARTMENT S	SQL2	INSERT	NO	NO
SYSTEM	DEPARTMENT S	SQL2	SELECT	NO	NO
SYSTEM	DEPARTMENT S	SQL2	UPDATE	NO	NO
SYSTEM	DEPARTMENT S	SQL2	REFERENCES	NO	NO
SYSTEM	DEPARTMENT S	SQL2	ON COMMIT REFRESH	NO	NO
SYSTEM	DEPARTMENT S	SQL2	QUERY REWR ITE	NO	NO
SYSTEM	DEPARTMENT S	SQL2	DEBUG	NO	NO

10 rows selected.

GRANTEE	TABLE_NAME	COLUMN_NAM	GRANTOR	PRIVILEGE	GRA
SYSTEM	EMPLOYEES	JOB_ID	SQL2	UPDATE	NO
SYSTEM	EMPLOYEES	SALARY	SQL2	UPDATE	NO

Practice C (continued)

- c. Produce a script to revoke the privileges. Save the output of the script into `my_file2.sql`. To save the file, select the **File** option for the output, and execute the code. Remember to save the file with a `.sql` extension. To execute the `my_file2.sql`, browse to locate the script, load the script, and execute the script.

'REVOKE' PRIVILEGE 'ON' TABLE_NAME 'FROM SYSTEM;'
REVOKE ALTER ON DEPARTMENTS FROM system;
REVOKE DELETE ON DEPARTMENTS FROM system;
REVOKE INDEX ON DEPARTMENTS FROM system;
REVOKE INSERT ON DEPARTMENTS FROM system;
REVOKE SELECT ON DEPARTMENTS FROM system;
REVOKE UPDATE ON DEPARTMENTS FROM system;
REVOKE REFERENCES ON DEPARTMENTS FROM system;
REVOKE ON COMMIT REFRESH ON DEPARTMENTS FROM system;
REVOKE QUERY REWRITE ON DEPARTMENTS FROM system;
REVOKE DEBUG ON DEPARTMENTS FROM system;

10 rows selected.

'REVOKE' PRIVILEGE 'ON' TABLE_NAME 'FROM SYSTEM;'
REVOKE UPDATE ON EMPLOYEES FROM system;



Oracle Architectural 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**

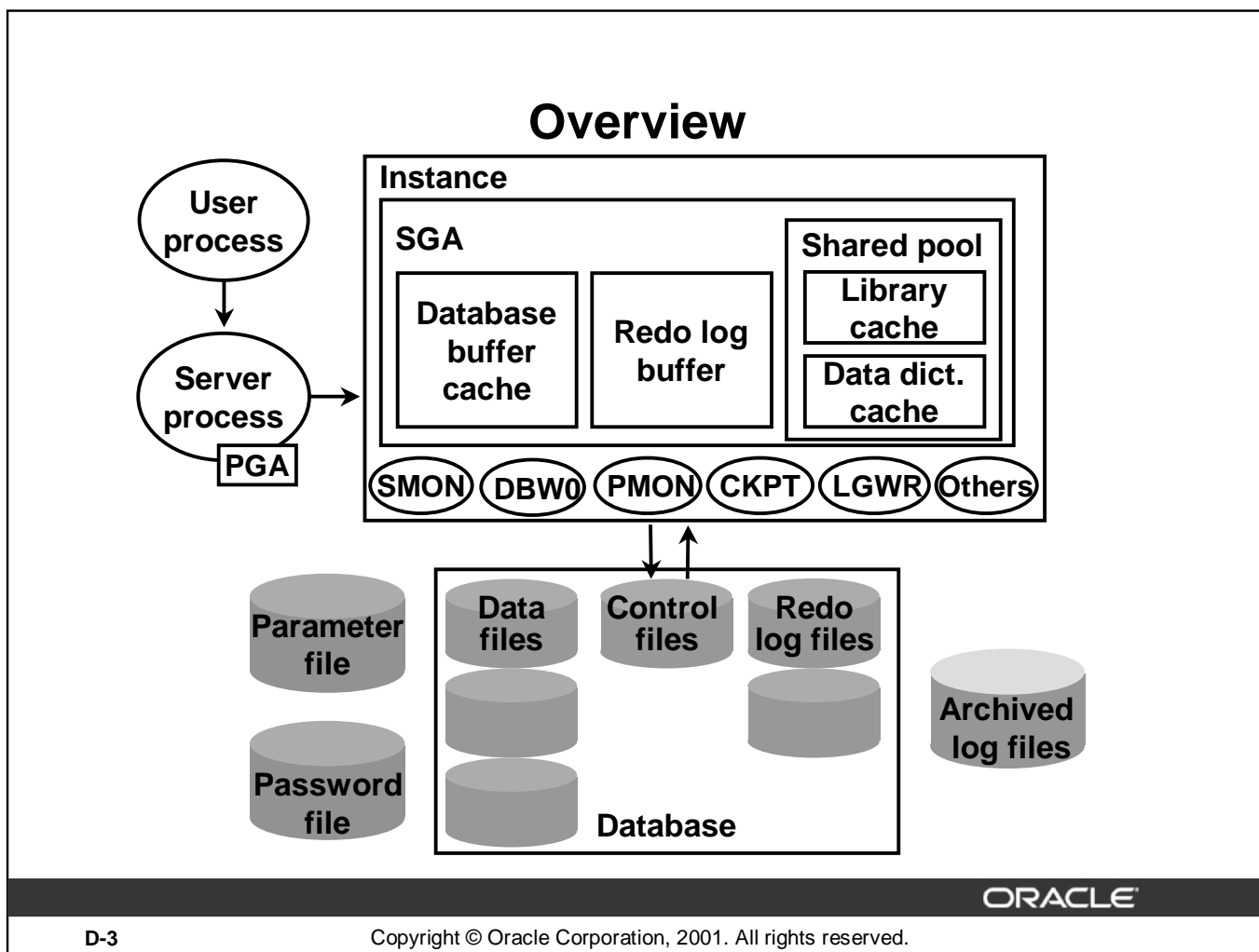
ORACLE

D-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

This appendix introduces Oracle Server architecture by describing the files, processes, and memory structures involved in establishing a database connection and executing a SQL command.



Overview

The Oracle Server is an object relational database management system that provides an open, comprehensive, integrated approach to information management.

Primary Components

There are several processes, memory structures, and files in an Oracle Server; however, not all of them are used when processing a SQL statement. Some are used to improve the performance of the database, ensure that the database can be recovered in the event of a software or hardware error, or perform other tasks necessary to maintain the database. The Oracle Server consists of an Oracle instance and an Oracle database.

Oracle Instance

An Oracle instance is the combination of the background processes and memory structures. The instance must be started to access the data in the database. Every time an instance is started, a system global area (SGA) is allocated and Oracle background processes are started. The SGA is a memory area used to store database information that is shared by database processes.

Background processes perform functions on behalf of the invoking process. They consolidate functions that would otherwise be handled by multiple Oracle programs running for each user. The background processes perform I/O and monitor other Oracle processes to provide increased parallelism for better performance and reliability.

Primary Components (continued)

Other Processes

The user process is the application program that originates SQL statements. The server process executes the SQL statements sent from the user process.

Database Files

Database files are operating system files that provide the actual physical storage for database information. The database files are used to ensure that the data is kept consistent and can be recovered in the event of a failure of the instance.

Other Files

Nondatabase files are used to configure the instance, authenticate privileged users, and recover the database in the event of a disk failure.

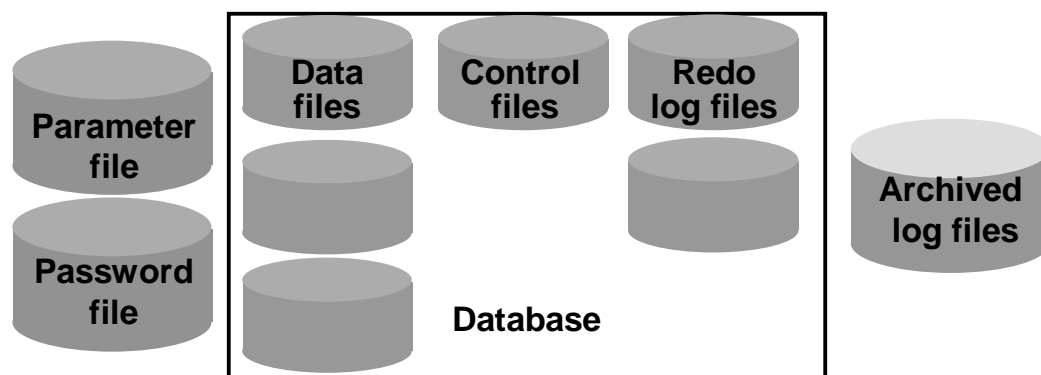
SQL Statement Processing

The user and server processes are the primary processes involved when a SQL statement is executed; however, other processes may help the server complete the processing of the SQL statement.

Oracle Database Administrators

Database administrators are responsible for maintaining the Oracle Server so that the server can process user requests. An understanding of the Oracle architecture is necessary to maintain it effectively.

Oracle Database Files



ORACLE

D-5

Copyright © Oracle Corporation, 2001. All rights reserved.

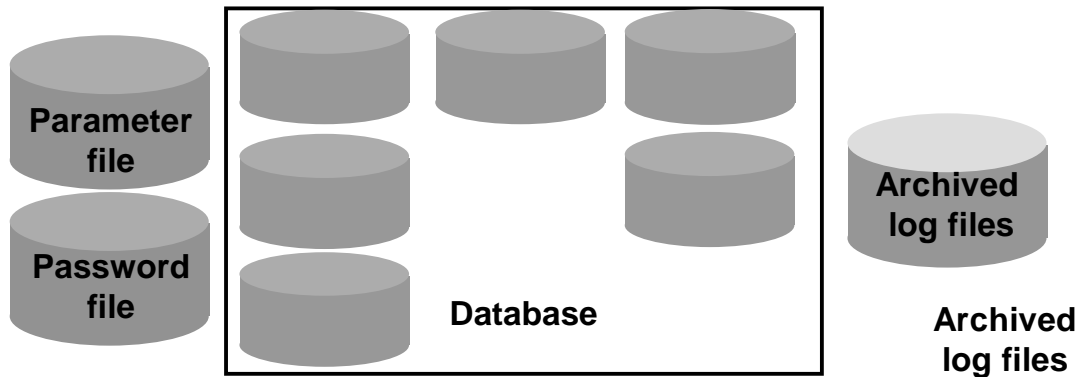
Oracle Database Files

An Oracle database is a collection of data that is treated as a unit. The general purpose of a database is to store and retrieve related information. The database has a logical structure and a physical structure. The physical structure of the database is the set of operating system files in the database. An Oracle database consists of three file types:

Data files contain the actual data in the database. The data is stored in user-defined tables, but data files also contain the data dictionary, before-images of modified data, indexes, and other types of structures. A database has at least one data file. The characteristics of data files are:

- A data file can be associated with only one database. Data files can have certain characteristics set so they can automatically extend when the database runs out of space. One or more data files form a logical unit of database storage called a tablespace. Redo logs contain a record of changes made to the database to enable recovery of the data in case of failures. A database requires at least two redo log files.
- Control files contain information necessary to maintain and verify database integrity. For example, a control file is used to identify the data files and redo log files. A database needs at least one control file.

Other Key Physical Structures



ORACLE

D-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Other Key Files

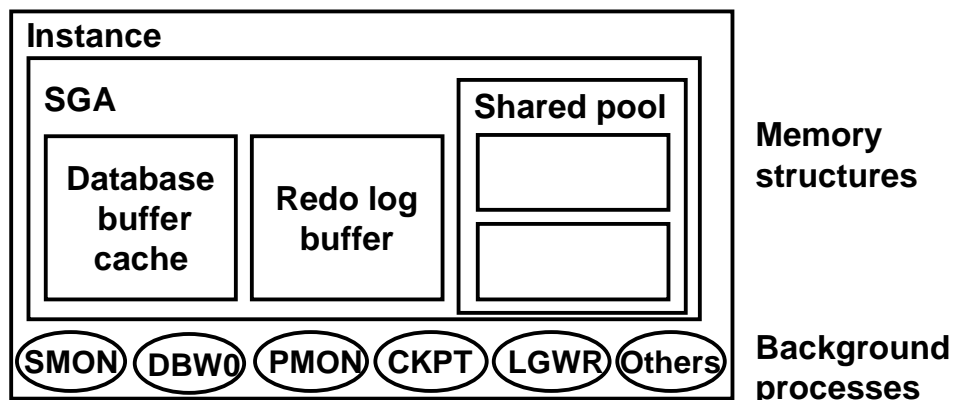
The Oracle Server also uses other files that are not part of the database:

- The parameter file defines the characteristics of an Oracle instance. For example, it contains parameters that size some of the memory structures in the SGA.
- The password file authenticates which users are permitted to start up and shut down an Oracle instance.
- Archived redo log files are offline copies of the redo log files that may be necessary to recover from media failures.

Oracle Instance

An Oracle instance:

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



ORACLE

D-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle Instance

An Oracle instance consists of the SGA memory structure and the background processes used to manage a database. An instance is identified by using methods specific to each operating system. The instance can open and use only one database at a time.

System Global Area

The SGA is a memory area used to store database information that is shared by database processes. It contains data and control information for the Oracle Server. It is allocated in the virtual memory of the computer where the Oracle server resides. The SGA consists of several memory structures:

- The shared pool is used to store the most recently executed SQL statements and the most recently used data from the data dictionary. These SQL statements may be submitted by a user process or, in the case of stored procedures, read from the data dictionary.
- The database buffer cache is used to store the most recently used data. The data is read from, and written to, the data files.
- The redo log buffer is used to track changes made to the database by the server and background processes.

System Global Area (continued)

The purpose of these structures is discussed in detail in later sections of this lesson.

There are also two optional memory structures in the SGA:

- Java pool: Used to store Java code
- Large pool: Used to store large memory structures not directly related to SQL statement processing; for example, data blocks copied during backup and restore operations

Background Processes

The background processes in an instance perform common functions that are needed to service requests from concurrent users without compromising the integrity and performance of the system. They consolidate functions that would otherwise be handled by multiple Oracle programs running for each user. The background processes perform I/O and monitor other Oracle processes to provide increased parallelism for better performance and reliability.

Depending on its configuration, an Oracle instance may include several background processes, but every instance includes these five required background processes:

- Database Writer (DBW0) is responsible for writing changed data from the database buffer cache to the data files.
- Log Writer (LGWR) writes changes registered in the redo log buffer to the redo log files.
- System Monitor (SMON) checks for consistency of the database and, if necessary, initiates recovery of the database when the database is opened.
- Process Monitor (PMON) cleans up resources if one of the Oracle processes fails.
- The Checkpoint Process (CKPT) is responsible for updating database status information in the control files and data files whenever changes in the buffer cache are permanently recorded in the database.

The following sections of this lesson explain how a server process uses some of the components of the Oracle instance and database to process SQL statements submitted by a user process.

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.**

ORACLE

D-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Components Used to Process SQL

Not all of the components of an Oracle instance are used to process SQL statements. The user and server processes are used to connect a user to an Oracle instance. These processes are not part of the Oracle instance, but are required to process a SQL statement.

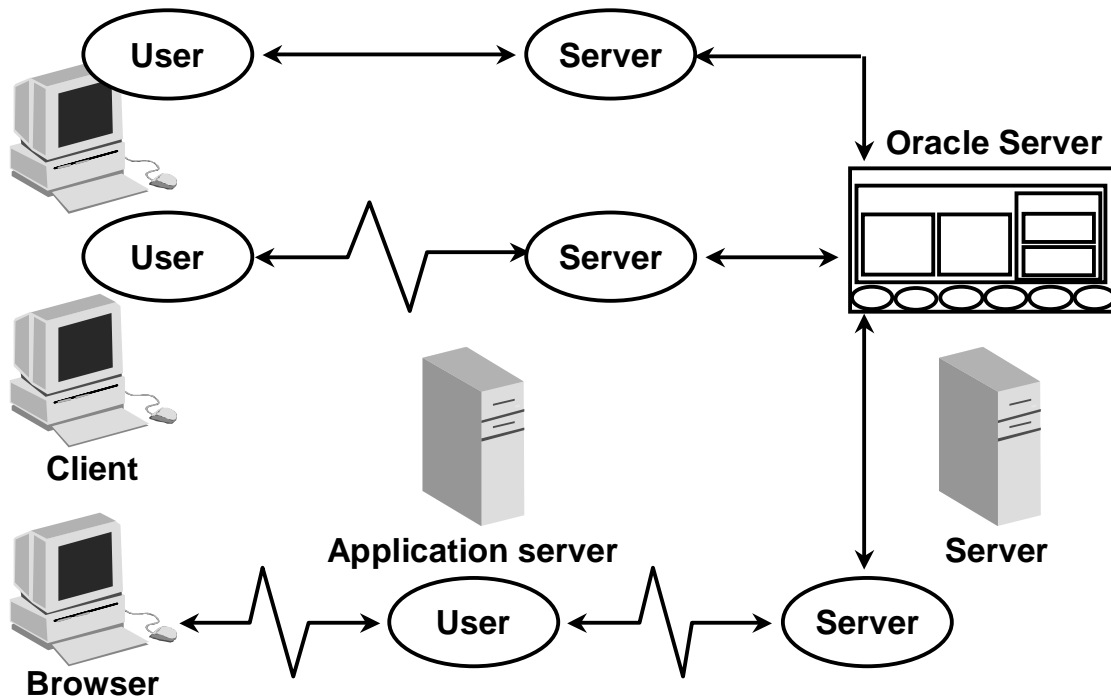
Some of the background processes, SGA structures, and database files are used to process SQL statements. Depending on the type of SQL statement, different components are used:

- Queries require additional processing to return rows to the user
- Data manipulation language (DML) statements require additional processing to log the changes made to the data
- Commit processing ensures that the modified data in a transaction can be recovered

Some required background processes do not directly participate in processing a SQL statement but are used to improve performance and to recover the database.

The optional background process, ARC0, is used to ensure that a production database can be recovered.

Connecting to an Instance



D-10

Copyright © Oracle Corporation, 2001. All rights reserved.

ORACLE

Processes Used to Connect to an Instance

Before users can submit SQL statements to the Oracle Server, they must connect to an instance.

The user starts a tool such as *iSQL*Plus* or runs an application developed using a tool such as Oracle Forms. This application or tool is executed in a *user process*.

In the most basic configuration, when a user logs on to the Oracle Server, a process is created on the computer running the Oracle Server. This process is called a server process. The server process communicates with the Oracle instance on behalf of the user process that runs on the client. The server process executes SQL statements on behalf of the user.

Connection

A connection is a communication pathway between a user process and an Oracle Server. A database user can connect to an Oracle Server in one of three ways:

- The user logs on to the operating system running the Oracle instance and starts an application or tool that accesses the database on that system. The communication pathway is established using the interprocess communication mechanisms available on the host operating system.

Processes Used to Connect to an Instance

Connection (continued)

- The user starts the application or tool on a local computer and connects over a network to the computer running the Oracle instance. In this configuration, called client-server, network software is used to communicate between the user and the Oracle Server.
- In a three-tiered connection, the user's computer communicates over the network to an application or a network server, which is connected through a network to the machine running the Oracle instance. For example, the user runs a browser on a network computer to use an application residing on an NT server that retrieves data from an Oracle database running on a UNIX host.

Sessions

A session is a specific connection of a user to an Oracle Server. The session starts when the user is validated by the Oracle Server, and it ends when the user logs out or when there is an abnormal termination. For a given database user, many concurrent sessions are possible if the user logs on from many tools, applications, or terminals at the same time. Except for some specialized database administration tools, starting a database session requires that the Oracle Server be available for use.

Note: The type of connection explained here, where there is a one-to-one correspondence between a user and server process, is called a dedicated server connection.

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**

ORACLE

D-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Query Processing Steps

Queries are different from other types of SQL statements because, if successful, they return data as results. Whereas other statements simply return success or failure, a query can return one row or thousands of rows.

There are three main stages in the processing of a query:

- Parse
- Execute
- Fetch

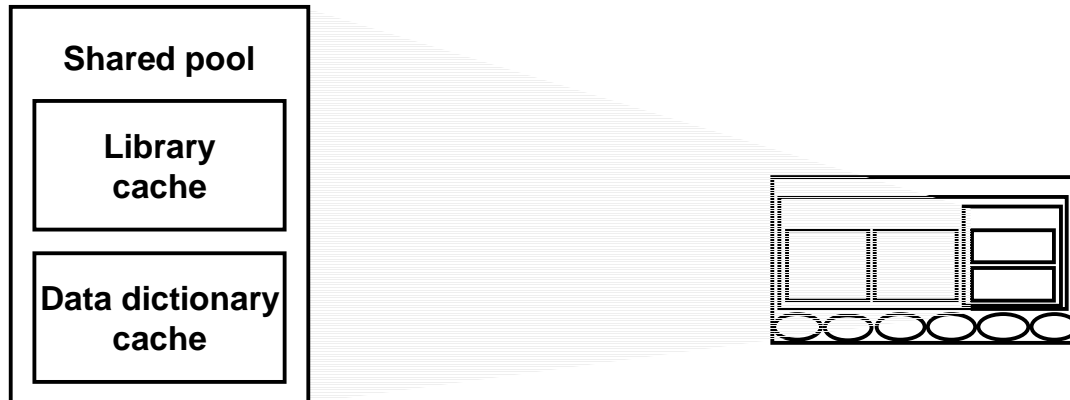
Parsing a SQL Statement

During the *parse* stage, the SQL statement is passed from the user process to the server process, and a parsed representation of the SQL statement is loaded into a shared SQL area.

During the parse, the server process performs the following functions:

- Searches for an existing copy of the SQL statement in the shared pool
- Validates the SQL statement by checking its syntax
- Performs data dictionary lookups to validate table and column definitions

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`.

ORACLE

D-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Shared Pool Components

During the parse stage, the server process uses the area in the SGA known as the shared pool to compile the SQL statement. The shared pool has two primary components:

- Library cache
- Data dictionary cache

Library Cache

The library cache stores information about the most recently used SQL statements in a memory structure called a shared SQL area. The shared SQL area contains:

- The text of the SQL statement
- The parse tree: A compiled version of the statement
- The execution plan: The steps to be taken when executing the statement

The optimizer is the function in the Oracle Server that determines the optimal execution plan.

If a SQL statement is reexecuted and a shared SQL area already contains the execution plan for the statement, the server process does not need to parse the statement. The library cache improves the performance of applications that reuse SQL statements by reducing parse time and memory requirements. If the SQL statement is not reused, it is eventually aged out of the library cache.

Shared Pool Components (continued)

Data Dictionary Cache

The data dictionary cache, also known as the dictionary cache or row cache, is a collection of the most recently used definitions in the database. It includes information about database files, tables, indexes, columns, users, privileges, and other database objects.

During the parse phase, the server process looks for the information in the dictionary cache to resolve the object names specified in the SQL statement and to validate the access privileges. If necessary, the server process initiates the loading of this information from the data files.

Sizing the Shared Pool

The size of the shared pool is specified by the initialization parameter `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`**

ORACLE

D-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Function of the Database Buffer Cache

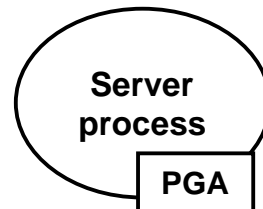
When a query is processed, the server process looks in the database buffer cache for any blocks it needs. If the block is not found in the database buffer cache, the server process reads the block from the data file and places a copy in the buffer cache. Because subsequent requests for the same block may find the block in memory, the requests may not require physical reads. The Oracle Server uses a least recently used algorithm to age out buffers that have not been accessed recently to make room for new blocks in the buffer cache.

Sizing the Database Buffer Cache

The size of each buffer in the buffer cache is equal to the size of an Oracle block, and it is specified by the `DB_BLOCK_SIZE` parameter. The number of buffers is equal to the value of the `DB_BLOCK_BUFFERS` parameter.

Program Global Area (PGA)

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



ORACLE

D-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Program Global Area Components

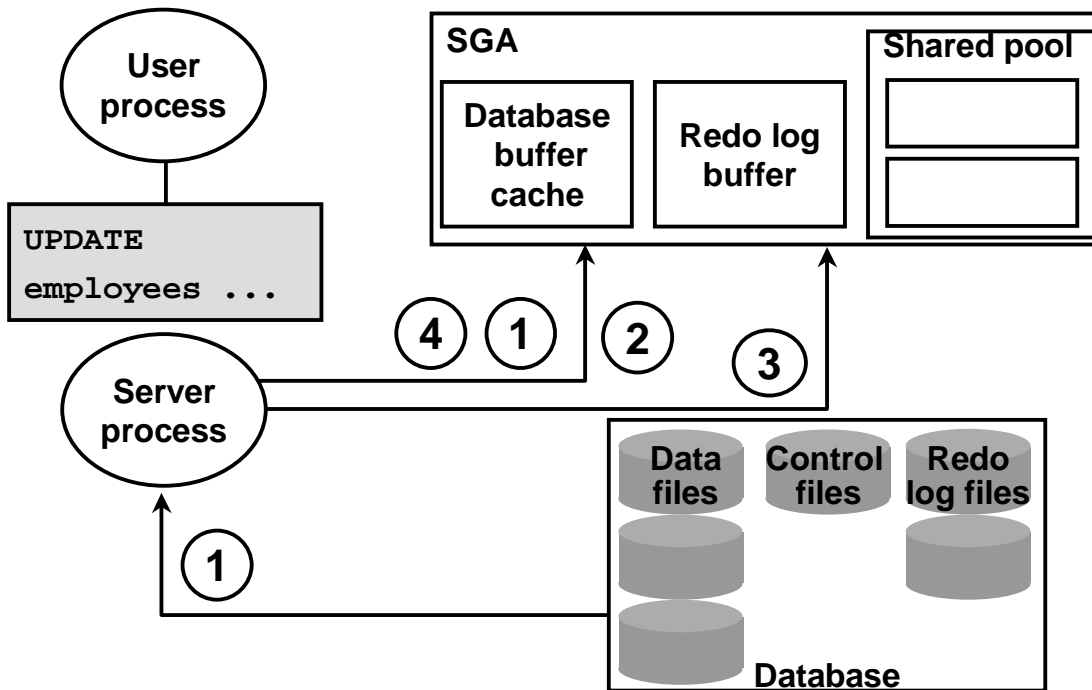
A program global area (PGA) is a memory region that contains data and control information for a server process. It is a nonshared memory created by Oracle when a server process is started. Access to it is exclusive to that server process and is read and written only by the Oracle Server code acting on behalf of it. The PGA memory allocated by each server process attached to an Oracle instance is referred to as the aggregated PGA memory allocated by the instance.

In a dedicated server configuration, the PGA of the server includes these components:

- **Sort area:** Used for any sorts that may be required to process the SQL statement
- **Session information:** Includes user privileges and performance statistics for the session
- **Cursor state:** Indicates the stage in the processing of the SQL statements that are currently used by the session
- **Stack space:** Contains other session variables

The PGA is allocated when a process is created and deallocated when the process is terminated.

Processing a DML Statement



ORACLE

D-17

Copyright © Oracle Corporation, 2001. All rights reserved.

DML Processing Steps

A data manipulation language (DML) statement requires only two phases of processing:

- Parse is the same as the parse phase used for processing a query
- Execute requires additional processing to make data changes

DML Execute Phase

To execute a DML statement:

- If the data and rollback blocks are not already in the buffer cache, the server process reads them from the data files into the buffer cache.
- The server process places locks on the rows that are to be modified.
- In the redo log buffer, the server process records the changes to be made to the rollback and data.
- The rollback block changes record the values of the data before it is modified. The rollback block is used to store the before image of the data, so that the DML statements can be rolled back if necessary.
- The data blocks changes record the new values of the data.

DML Processing Steps

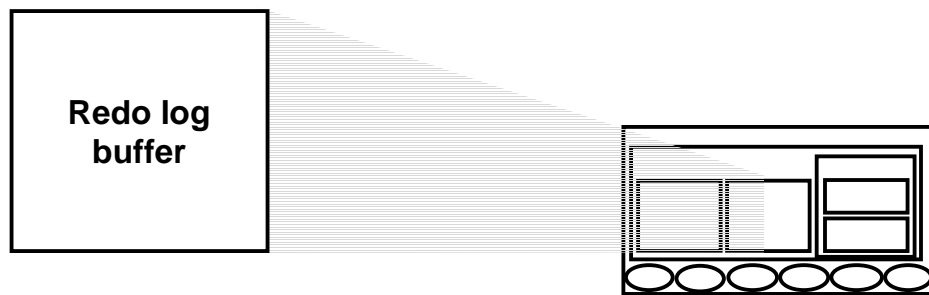
DML Execute Phase (continued)

The server process records the before image to the rollback block and updates the data block. Both of these changes are done in the database buffer cache. Any changed blocks in the buffer cache are marked as dirty buffers: that is, buffers that are not the same as the corresponding blocks on the disk.

The processing of a DELETE or INSERT command uses similar steps. The before image for a DELETE contains the column values in the deleted row, and the before image of an INSERT contains the row location information.

Because the changes made to the blocks are only recorded in memory structures and are not written immediately to disk, a computer failure that causes the loss of the SGA can also lose these changes.

Redo Log Buffer



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

ORACLE

D-19

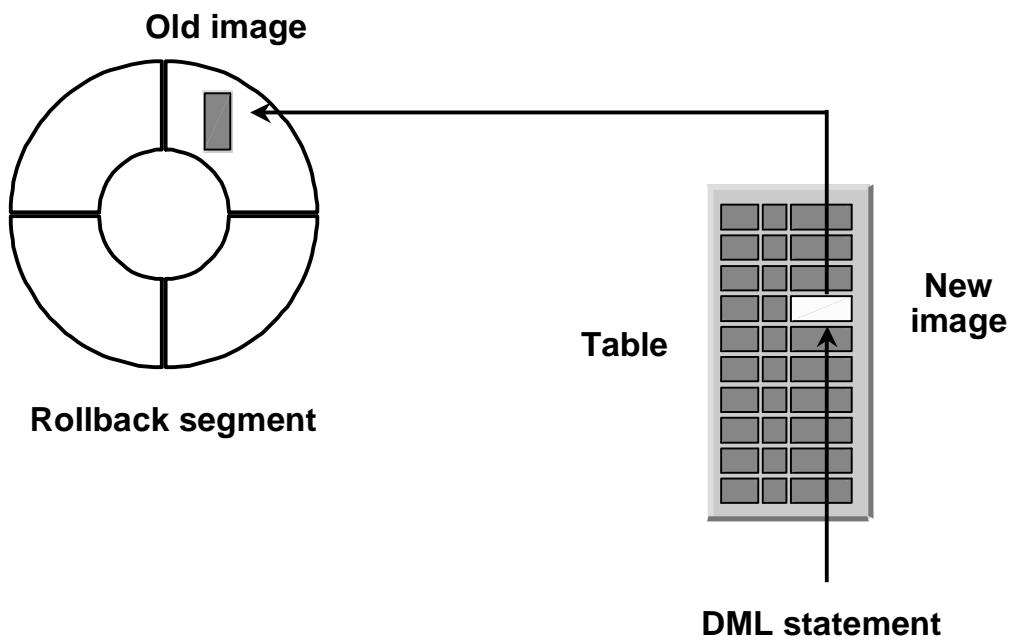
Copyright © Oracle Corporation, 2001. All rights reserved.

Redo Log Buffer Characteristics

The server process records most of the changes made to data file blocks in the redo log buffer, which is a part of the SGA. The redo log buffer has the following characteristics:

- Its size in bytes is defined by the LOG_BUFFER parameter.
- It records the block that is changed, the location of the change, and the new value in a redo entry. A redo entry makes no distinction between the type of block that is changed; it simply records which bytes are changed in the block.
- The redo log buffer is used sequentially, and changes made by one transaction may be interleaved with changes made by other transactions.
- It is a circular buffer that is reused after it is filled, but only after all the old redo entries are recorded in the redo log files.

Rollback Segment



ORACLE

D-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Rollback Segment

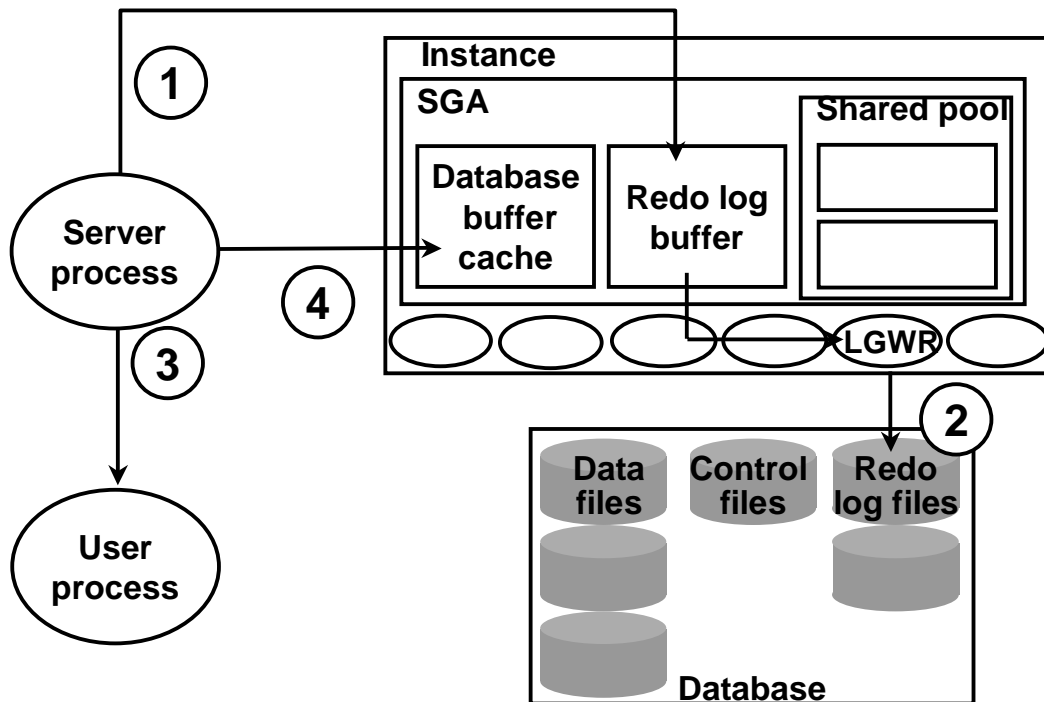
Before making a change, the server process saves the old data value into a rollback segment. This before image is used to:

- Undo the changes if the transaction is rolled back
- Provide read consistency by ensuring that other transactions do not see uncommitted changes made by the DML statement
- Recover the database to a consistent state in case of failures

Rollback segments, like tables and indexes, exist in data files, and rollback blocks are brought into the database buffer cache as required. Rollback segments are created by the DBA.

Changes to rollback segments are recorded in the redo log buffer.

COMMIT Processing



ORACLE

D-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Fast COMMIT

The Oracle Server uses a fast commit mechanism that guarantees that the committed changes can be recovered in case of instance failure.

System Change Number

Whenever a transaction commits, the Oracle Server assigns a commit system change number (SCN) to the transaction. The SCN is monotonically incremented and is unique within the database. It is used by the Oracle Server as an internal time stamp to synchronize data and to provide read consistency when data is retrieved from the data files. Using the SCN enables the Oracle Server to perform consistency checks without depending on the date and time of the operating system.

Steps in Processing COMMITs

When a COMMIT is issued, the following steps are performed:

- The server process places a commit record, along with the SCN, in the redo log buffer.
- LGWR performs a contiguous write of all the redo log buffer entries up to and including the commit record to the redo log files. After this point, the Oracle Server can guarantee that the changes will not be lost even if there is an instance failure.

Fast COMMIT

Steps in Processing COMMITs (continued)

- The user is informed that the COMMIT is complete.
- The server process records information to indicate that the transaction is complete and that resource locks can be released.

Flushing of the dirty buffers to the data file is performed independently by DBW0 and can occur either before or after the commit.

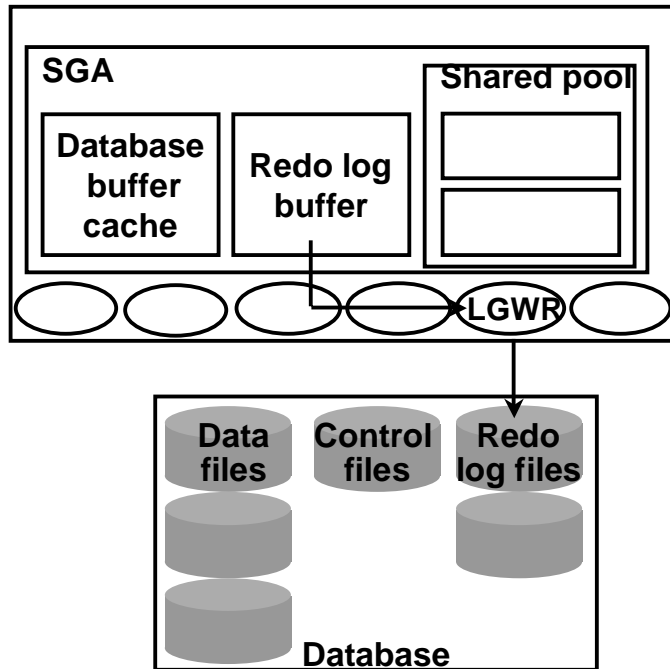
Advantages of the Fast COMMIT

The fast commit mechanism ensures data recovery by writing changes to the redo log buffer instead of the data files. It has the following advantages:

- Sequential writes to the log files are faster than writing to different blocks in the data file.
- Only the minimal information that is necessary to record changes is written to the log files, whereas writing to the data files would require whole blocks of data to be written.
- If multiple transactions request to commit at the same time, the instance piggybacks redo log records into a single write.
- Unless the redo log buffer is particularly full, only one synchronous write is required per transaction. If piggybacking occurs, there can be less than one synchronous write per transaction.
- Because the redo log buffer may be flushed before the COMMIT, the size of the transaction does not affect the amount of time needed for an actual COMMIT operation.

Note: Rolling back a transaction does not trigger LGWR to write to disk. The Oracle Server always rolls back uncommitted changes when recovering from failures. If there is a failure after a rollback, before the rollback entries are recorded on disk, the absence of a commit record is sufficient to ensure that the changes made by the transaction are rolled back.

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**

ORACLE

D-23

Copyright © Oracle Corporation, 2001. All rights reserved.

LOG Writer

LGWR performs sequential writes from the redo log buffer to the redo log file under the following situations:

- When a transaction commits
- When the redo log buffer is one-third full
- When there is more than a megabyte of changes recorded in the redo log buffer
- Before DBW0 writes modified blocks in the database buffer cache to the data files

Because the redo is needed for recovery, LGWR confirms the COMMIT only after the redo is written to disk.

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**

ORACLE

D-24

Copyright © Oracle Corporation, 2001. All rights reserved.

Other Required Processes

Four other required processes do not participate directly in processing SQL statements:

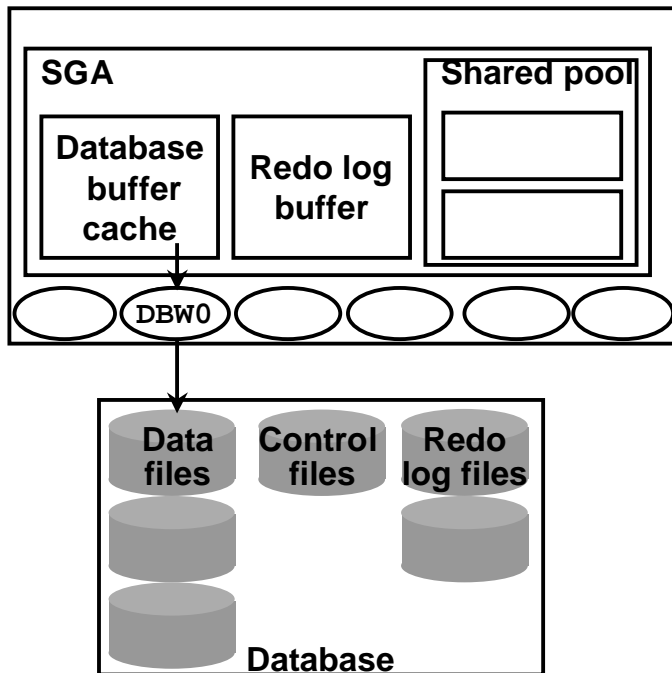
- Database Writer (DBW0)
- Process Monitor (PMON)
- System Monitor (SMON)
- Checkpoint (CKPT)

The checkpoint process is used to synchronize database files.

The Archiver Process

All other background processes are optional, depending on the configuration of the database; however, one of them, ARC0, is crucial to recovering a database after the loss of a disk. The ARC0 process 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

ORACLE

D-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Database Writer

The server process records changes to rollback and data blocks in the buffer cache. The Database Writer (DBW0) writes the dirty buffers from the database buffer cache to the data files. It ensures that a sufficient number of free buffers (buffers that can be overwritten when server processes need to read in blocks from the data files) are available in the database buffer cache. Database performance is improved because server processes make changes only in the buffer cache, and the DBW0 defers writing to the data files until one of the following events occurs:

- The number of dirty buffers reaches a threshold value
- A process scans a specified number of blocks when scanning for free buffers and cannot find any
- A timeout occurs (every three seconds)
- A checkpoint occurs (a checkpoint is a means of synchronizing the database buffer cache with the data file)

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**

ORACLE

D-26

Copyright © Oracle Corporation, 2001. All rights reserved.

SMON: System Monitor

If the Oracle instance fails, any information in the SGA that has not been written to disk is lost. For example, the failure of the operating system causes an instance failure. After the loss of the instance, the background process SMON automatically performs instance recovery when the database is reopened. Instance recovery consists of the following steps:

- Rolling forward to recover data that has not been recorded in the data files but that has been recorded in the online redo log. This data has not been written to disk because of the loss of the SGA during instance failure. During this process, SMON reads the redo log files and applies the changes recorded in the redo log to the data blocks. Because all committed transaction have been written to the redo logs, this process completely recovers these transactions.
- Opening the database so users can log on. Any data that is not locked by unrecovered transactions is immediately available.
- Rolling back uncommitted transactions. They are rolled back by SMON or by the individual server processes as they access locked data.

SMON also performs some space maintenance functions:

- It combines, or coalesces, adjacent areas of free space in the data files.
- It deallocates temporary segments to return them as free space in data files. Temporary segments are used to store data during SQL statement processing.

PMON: Process Monitor

Cleans up after failed processes by:

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

ORACLE

D-27

Copyright © Oracle Corporation, 2001. All rights reserved.

PMON Functionality

The background process PMON cleans up after failed processes by:

- Rolling back the user's current transaction
- Releasing all currently held table or row locks
- Freeing other resources currently reserved by the user

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**

ORACLE

D-28

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

The Oracle database includes these files:

- **Control files:** Contain information required to verify the integrity of the database, including the names of the other files in the database (The control files are usually mirrored.)
- **Data files:** Contain the data in the database, including tables, indexes, rollback segments, and temporary segments
- **Online redo logs:** Contain the changes made to the data files (Online redo logs are used for recovery and are usually mirrored.)

Other files commonly used with the database include:

- **Parameter file:** Defines the characteristics of an Oracle instance
- **Password file:** Authenticates privileged database users
- **Archived redo logs:** Are backups of the online redo logs

Summary

SGA Memory Structures

The System Global Area (SGA) has three primary structures:

- Shared pool: Stores the most recently executed SQL statements and the most recently used data from the data dictionary
- Database buffer cache: Stores the most recently used data
- Redo log buffer: Records changes made to the database using the instance

Background Processes

A production Oracle instance includes these processes:

- Database Writer (DBW0): Writes changed data to the data files
- Log Writer (LGWR): Records changes to the data files in the online redo log files
- System Monitor (SMON): Checks for consistency and initiates recovery of the database when the database is opened
- Process Monitor (PMON): Cleans up the resources if one of the processes fails
- Checkpoint Process (CKPT): Updates the database status information after a checkpoint
- Archiver (ARC0): Backs up the online redo log to ensure recovery after a media failure (This process is optional, but is usually included in a production instance.)

Depending on its configuration, the instance may also include other processes.

SQL Statement Processing Steps

The steps used to process a SQL statement include:

- Parse: Compiles the SQL statement
- Execute: Identifies selected rows or applies DML changes to the data
- Fetch: Returns the rows queried by a `SELECT` statement

Index

A

ACCESS PARAMETER 6-19

ALL INSERT 6-7

C

CASE 4-12

child node 5-10

Comparison operator 4-4

composite column 3-17

Concatenated grouping 3-21

conditional INSERT 6-7,6-13,6-14

CONNECT BY 5-5,5-7,5-13

correlated subquery 4-2,4-14,4-15

correlated subquery to delete 4-24

correlated subquery to update row 4-21

correlated update 4-22

correlation 4-17

CREATE DATABASE 2-11

CREATE INDEX 6-24

cross-tabular reports 3-9

cross-tabulation rows 3-6

cross-tabulation values 3-10

CUBE 3-2,3-6,3-9

CURRENT_DATE 2-8

CURRENT_TIMESTAMP 2-9

D

datetime function 2-2

daylight saving 2-5

DBTIMEZONE 2-11

DEFAULT DIRECTORY 6-19

duplicate row 1-11

E

EXIST 4-18,4-19

external table 6-2,6-18

EXTRACT 2-12

expression 2-12, 4-12

F

FIRST INSERT 6-7,6-13,6-14

function 2-6,2-10, 2-11, 2-13, 2-14, 2-15

G

Greenwich Mean Time 2-3

GROUP BY 3-3, 3-4

GROUP BY ROLLUP 3-17

Group date 3-2

GROUPING 3-11,3-12

GROUPING SET 3-13

H

HAVING 3-5

I

inner query 4-5

INSERT 6-2

INTERSECT 1-12

L

LEVEL 5-10

LOCALTIMESTAMP 2-10

LOCATION 6-19

LPAD 5-11

O

operator 1-2,1-8,1-11,3-2,3-8,4-18

P

pseudocolumn 5-10

pairwise comparisons 4-7

parent-child relationship 5-4

Pivoting 6-7

PRIOR 5-7

prune the tree 5-13

M

MINUS 1-14

multiple-column subqueries 4-2,4-6,4-8

N

nonpairwise comparisons 4-7

NOT EXIST 4-20

NOT IN 4-20

O

ORDER BY 1-20

ORGANIZATION EXTERNAL 6-18

outer query 4-5

R

REJECT LIMIT 6-19

ROLLUP 3-2,3-6,3-7,3-8

root node 5-10

row 3-8

S

scalar subquery 4-11

SET 1-2,1-3

SESSIONTIMEZONE 2-11

SET TIME_ZONE 2-11

START WITH 5-5,5-6

superaggregate rows 3-9

T

time zone 2-3

TIMESTAMP WITH LOCAL TIME ZONE 2-5

TIMEZONE_ABBR 2-12

TIMEZONE_REGION 2-12

TO_TIMESTAMP 2-14

TO_YMINTERVAL 2-15

tree-structured report 5-2

TYPE access_driver_type 6-19

TZ_OFFSET 2-6

U

Unconditional INSERT 6-7

UNION 1-7,1-11

UNION ALL 1-10

V

V\$TIMEZONE_NAME 2-12

W

WITH clause 4-2

