Reporting Aggregated Data Using the Group Functions

Objectives

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

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

What Are Group Functions?

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

EMPLOYEES

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

Types of Group Functions

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

Group Functions: Syntax

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

Using the AVG and SUM Functions

You can use AVG and SUM for numeric data.

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

Using the MIN and MAX Functions

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

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

Using the COUNT Function

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

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

COUNT(*expr) returns the number of rows with nonnull values for the expr:*

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

Using the DISTINCT Keyword

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

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

Group Functions and Null Values

Group functions ignore null values in the column:

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

The NVL function forces group functions to include null values:

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

Creating Groups of Data: GROUP BY Clause Syntax

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

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

Using the GROUP BY Clause

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

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

Using the GROUP BY Clause

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

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

Using the GROUP BY Clause on Multiple Columns

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

Illegal Queries Using Group Functions

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

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

Column missing in the GROUP BY clause

Illegal Queries Using Group Functions

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

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

Cannot use the WHERE clause to restrict groups

Restricting Group Results with the HAVING Clause

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

1. Rows are grouped.

2. The group function is applied.

3. Groups matching the HAVING clause are displayed.

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

Using the HAVING Clause

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

Using the HAVING Clause

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

Nesting Group Functions

Display the maximum average salary:

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

Displaying Data from Multiple Tables

Objectives

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

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

Types of Joins

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

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

Joining Tables Using SQL:1999 Syntax

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

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

Creating Natural Joins

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

Retrieving Records with Natural Joins

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

Creating Joins with the USING Clause

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

Retrieving Records with the USING Clause

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

Qualifying Ambiguous Column Names

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

Using Table Aliases

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

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

Creating Joins with the ON Clause

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

Retrieving Records with the ON Clause

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

Self-Joins Using the ON Clause

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

Applying Additional Conditions to a Join

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

Creating Three-Way Joins with the ON Clause

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

Retrieving Records with Non-Equijoins

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

INNER Versus OUTER Joins

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

LEFT OUTER JOIN

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

RIGHT OUTER JOIN

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

FULL OUTER JOIN

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

Cartesian Products

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

```
select last_name, department_name from
employees, departments;
```

Generating a Cartesian Product

EMPLOYEES (20 rows)

DEPARTMENTS (8 rows)

Cartesian product:
20 x 8 = 160 rows

Creating Cross Joins

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

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