Oracle 11g - SQL

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 Type 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
1	10	4400
2	20	13000
3	20	6000
4	110	12000
5	110	8300
6	90	24000
7	90	17000
8	90	17000
9	60	9000
10	60	6000
11	60	4200
12	50	5800
13	50	3500
14	50	3100
15	50	2600

...

Maximum salary in EMPLOYEES table

	MAX(SALARY)
1	24000

Types of Group Functions

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

Group Functions: Syntax

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

Using the AVG and SUM Functions

You can use AVG and SUM for numeric data.

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

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

Using the MIN and MAX Functions

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

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

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

Using the COUNT Function

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

1

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

	COUNT(*)
1	5

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

2

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

	COUNT(COMMISSION_PCT)
1	3

Using the DISTINCT Keyword

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

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

	COUNT(DISTINCTDEPARTMENT_ID)
1	7

Group Functions and Null Values

Group functions ignore null values in the column:

1

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

	AVG(COMMISSION_PCT)
1	0.2125

The NVL function forces group functions to include null values:

2

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

	AVG(NVL(COMMISSION_PCT,0))
1	0.0425

Creating Groups of Data

EMPLOYEES

	DEPARTMENT_ID	SALARY	
1	10	4400	4400
2	20	13000	9500
3	20	6000	
4	50	2500	
5	50	2600	
6	50	3100	3500
7	50	3500	
8	50	5800	
9	60	9000	
10	60	6000	6400
11	60	4200	
12	80	11000	
13	80	8600	10033
14	80	10500	
15	90	17000	
16	90	24000	19333
17	90	17000	
18	110	8300	10150
19	110	12000	
20	(null)	7000	7000

Average salary in the EMPLOYEES table for each department

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

Creating Groups of Data: GROUP BY Clause Syntax

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

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

Using the GROUP BY Clause

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

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

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

Using the GROUP BY Clause on Multiple Columns

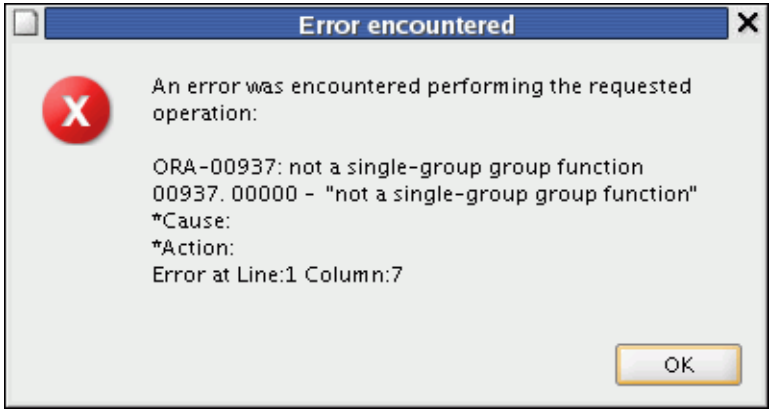
```
SELECT    department_id dept_id, job_id, SUM(salary)
FROM      employees
GROUP BY department_id, job_id ;
```

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

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;
```

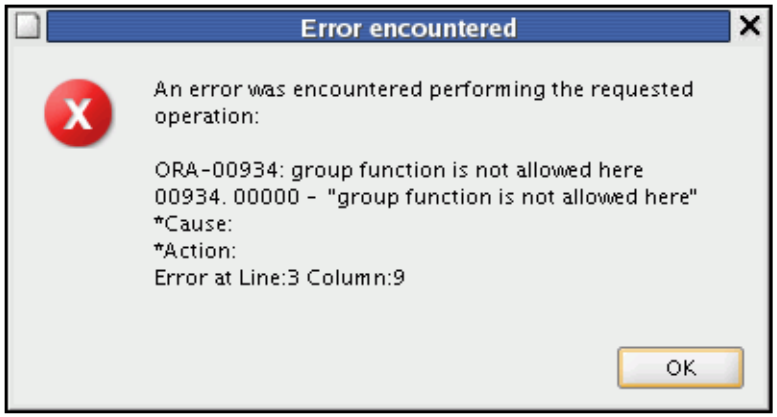


Column missing in the GROUP BY clause

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

EMPLOYEES

	DEPARTMENT_ID	SALARY
1	10	4400
2	20	13000
3	20	6000
4	110	12000
5	110	8300
6	90	24000
7	90	17000
8	90	17000
9	60	9000
10	60	6000
11	60	4200
12	50	5800
13	50	3500
14	50	3100
15	50	2600

...

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

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

Restricting Group Results with the HAVING Clause

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

- Rows are grouped.
- The group function is applied.
- Groups matching the HAVING clause are displayed.

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

Using the HAVING Clause

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

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

Using the HAVING Clause

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

	JOB_ID	PAYROLL
1	IT_PROG	19200
2	AD_PRES	24000
3	AD_VP	34000

Nesting Group Functions

Display the maximum average salary:

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

	MAX(AVG(SALARY))
1	19333.3333333333333333333333333333333333

Oracle Analytical functions

Grouping & Aggregate Enhancement

AGGREGATION ENHANCEMENTS

- **Grouping and Aggregating Data Using SQL**

Overview

- To improve aggregation performance in a data warehouse, Oracle Database provides the following functionality:

- **CUBE** and **ROLLUP** extensions to the GROUP BY clause
- **GROUPING SETS** expression
- **GROUPING** function

AGGREGATION ENHANCEMENTS

- To leverage the power of the database server, the SQL engine should offer powerful aggregation commands
- Oracle's extensions to SQL's GROUP BY clause provide this power futures:
  - Quicker and more efficient query processing
  - Reduced client processing loads and network traffic: **aggregation work** is **shifted** to **servers**
  - Simplified programming: less SQL code needed
  - Opportunities for caching aggregations: similar queries can leverage prior work
- **Without** the aggregation **enhancements**, many aggregation tasks require **multiple queries** against the same tables.

Using the ROLLUP and CUBE Operators

Using the ROLL UP operator

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

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	10	AD_ASST	4400
2	10		4400
3	20	MK_MAN	13000
4	20	MK_REP	6000
5	20		19000
6	30	PU_MAN	11000
7	30	PU_CLERK	13900
8	30		24900
9	40	HR_REP	6500
10	40		6500
11	50	ST_MAN	36400
12	50	SH_CLERK	64300
13	50	ST_CLERK	55700
14	50		156400
15			211200

1. Total by DEPARTMENT_ID and JOB_ID
2. Total by DEPARTMENT_ID
3. Grand total

Using the CUBE operator

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

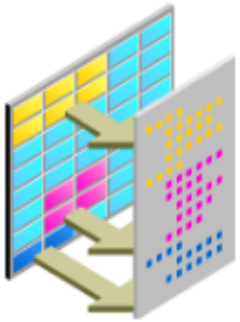
	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1			211200
2		HR_REP	6500
3		MK_MAN	13000
4		MK_REP	6000
5		PU_MAN	11000
6		ST_MAN	36400
7		AD_ASST	4400
8		PU_CLERK	13900
9		SH_CLERK	64300
10		ST_CLERK	55700
11	10		4400
12	10	AD_ASST	4400
13	20		19000
14	20	MK_MAN	13000
15	20	MK_REP	6000
16	30		24900
17	30	PU_MAN	11000
18	30	PU_CLERK	13900

1. Grand total
2. Total by JOB_ID
3. Total by DEPARTMENT_ID and JOB_ID
4. Total by DEPARTMENT_ID

Grouping Sets

Overview

- The `GROUPING SETS` syntax is used to define multiple groupings in the same query.
- Grouping set efficiency:
  - Only one pass over the base table is required.
  - There is no need to write complex `UNION` statements.
  - The more elements `GROUPING SETS` has, the greater the performance benefit.



Example of Grouping Sets

Comparison

Without `GROUPING SETS` expression

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

With `GROUPING SETS` expression

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

	DEPARTMENT_ID	JOB_ID	MANAGER_ID	AVG(SALARY)
1		SH_CLERK	122	3200
2		AC_MGR	101	12000
3		ST_MAN	100	7280
4		ST_CLERK	121	2675
5		SA_REP	148	8650

2

	DEPARTMENT_ID	JOB_ID	MANAGER_ID	AVG(SALARY)
33	110	AC_ACCOUNT		8300
34	90	AD_VP		17000
35	50	ST_CLERK		2785

1

53				6461.831775700934579439252336448598130841

3

The Grouping Function

Explanation with Example

The GROUPING function:

- Is used with the CUBE or ROLLUP operator
- Is used to find the groups forming the subtotal in a row
- Returns 0 or 1

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

	DEPTID	JOB	SUM(SALARY)	GRP_DEPT	GRP_JOB
1	10	AD_ASST	4400	0	0
2	10		4400	0	1
3	20	MK_MAN	13000	0	0
4	20	MK_REP	6000	0	0
5	20		19000	0	1
6	30	PU_MAN	11000	0	0
7	30	PU_CLERK	13900	0	0
8	30		24900	0	1
9	40	HR_REP	6500	0	0
10	40		6500	0	1
11			54800	1	1

1

2

3

Advantages on Group by enhance

- The extensions to the GROUP BY clause, allow the user to specify exactly what aggregations are needed in a single query
- The GROUP BY extensions enable subtotal and grand total calculations. They allow us following:

- GROUPING SETS - perform multiple independent groupings for just the subtotals needed.
- CUBE and ROLLUP – specify complex grouping sets with efficient and convenient shortcuts
- Composite Columns - skip unneeded aggregation levels
- Concatenated groupings - concisely specify many complex groupings by automatically generating needed combinations

Scenario Based Reference

- The feature descriptions begins with a brief **reference scenario**. GROUPING SETS presented, followed by ROLLUP and CUBE.
- We build on these concepts with discussions of composite columns and concatenated grouping sets.
- Finally we cover hierarchical cubes and grouping functions, a features supporting OLAP tasks.

Reference Scenario:

- To illustrate the concepts of GROUP BY extensions, this section uses a hypothetical ***Videotape* sales** and **rental** company.
- Check the **File**: "Scenario base Group Enhancement"