

# 1

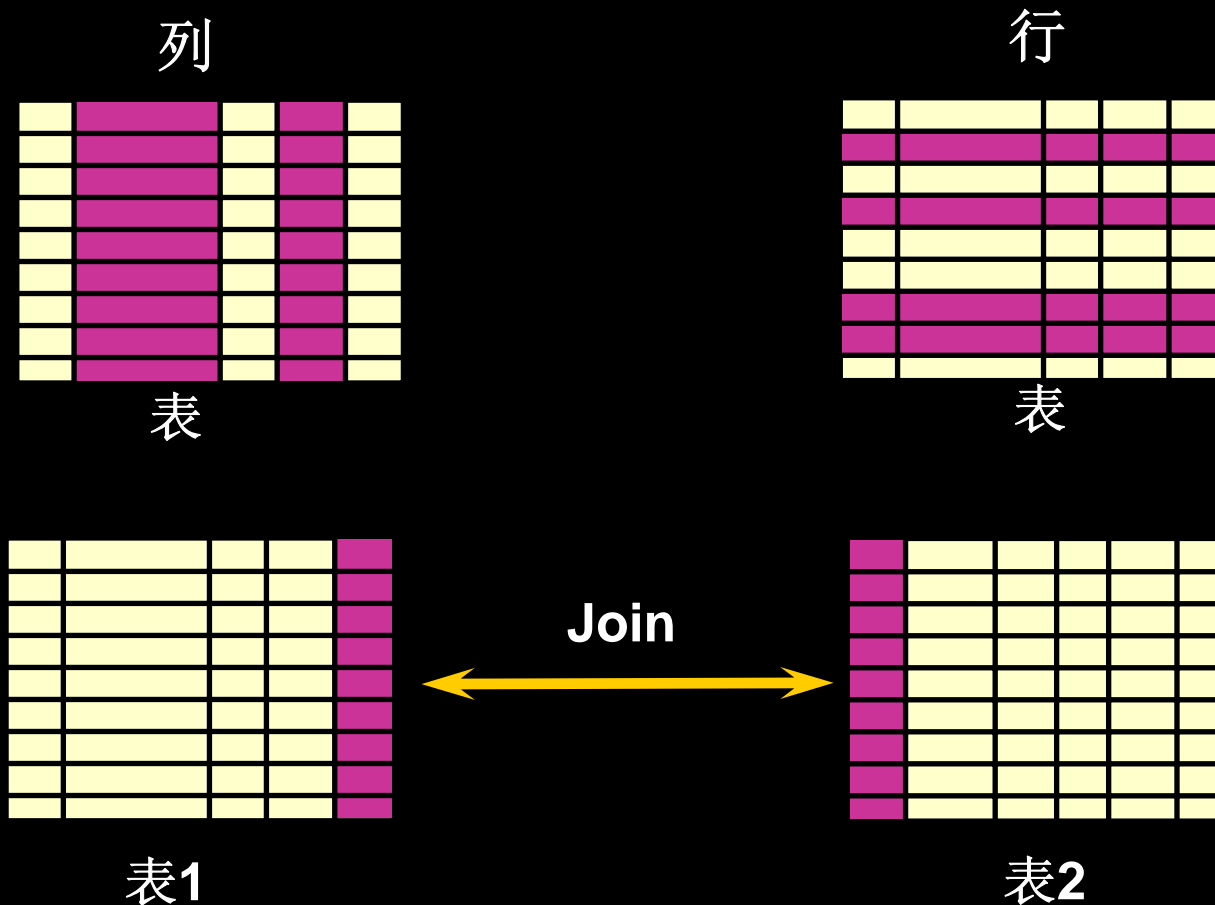
## 基本SQL SELECT语句

# 目标

通过本章学习，您将可以：

- 列举 **SQL SELECT**语句的功能。
- 执行简单的选择语句。
- **SQL** 语言和 **SQL\*Plus** 命令的不同。

# SQL SELECT 语句的功能



# 基本 SELECT 语句

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

- **SELECT** 标识 选择哪些列。
- **FROM** 标识从哪个表中选择。

# 选择全部列

```
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.

# 选择特定的列

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

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

8 rows selected.

# Writing SQL Statements

- **SQL** 语言大小写不敏感。
- **SQL** 可以写在一行或者多行
- 关键字不能被缩写也不能分行
- 各子句一般要分行写。
- 使用缩进提高语句的可读性。

# 列头设置

- **SQL\*Plus:**
  - 字符和日期类型的列左对齐
  - 字符类型的列右对齐
  - 默认头显示方式:大写



# 算术运算符

数字和日期使用的数学表达式。

操作符	描述
+	加
-	减
*	乘
/	除

# 使用数学运算符

```
SELECT last_name, salary, salary + 300
FROM   employees;
```

LAST_NAME	SALARY	SALARY+300
King	24000	24300
Kochhar	17000	17300
De Haan	17000	17300
Hunold	9000	9300
Ernst	6000	6300
...		
Hartstein	13000	13300
Fay	6000	6300
Higgins	12000	12300
Gietz	8300	8600

20 rows selected.

# 操作符优先级



- 乘除的优先级高于加减。
- 同一优先级运算符从左向右执行。
- 括号内的运算先执行。

# 操作符优先级

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

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

■ ■ ■

Hartstein	13000	156100
Fay	6000	72100
Higgins	12000	144100
Gietz	8300	99700

20 rows selected.

# 使用括号

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

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

...

Hartstein	13000	157200
Fay	6000	73200
Higgins	12000	145200
Gietz	8300	100800

20 rows selected.

# 定义空值

- 空值是无效的，未指定的，未知的或不可预知的值。
- 空值不是空格或者0。

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

LAST_NAME	JOB_ID	SALARY	COMMISSION_PCT
King	AD_PRES	24000	
Kochhar	AD_VP	17000	

...

Zlotkey	SA_MAN	10500	.2
Abel	SA_REP	11000	.3
Taylor	SA_REP	8600	.2

...

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

20 rows selected.

# 空值在数学运算中的使用

包含空值的数学表达式的值都为空值

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

Kochhar	
King	
LAST_NAME	12*SALARY*COMMISSION_PCT
...	
Zlotkey	25200
Abel	39600
Taylor	20640
...	
Gietz	

20 rows selected.

# 列的别名

列的别名:

- 重命名一个列。
- 便于计算。
- 紧跟列名，也可以在列名和别名之间加入关键字‘**AS**’，以便在别名中包含空格或特殊的字符并区分大小写。



# 使用别名

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

NAME	COMM
King	
Kochhar	
De Haan	

...

20 rows selected.

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

Name	Annual Salary
King	288000
Kochhar	204000
De Haan	204000

...

20 rows selected.

# 连接符

连接符:

- 把列与列，列与字符连接在一起。
- 用 ‘||’表示。
- 可以用来‘合成’列。

# 连接符应用举例

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

Employees	
KingAD_PRES	
KochharAD_VP	
De HaanAD_VP	
HunoldIT_PROG	
ErnstIT_PROG	
LorentzIT_PROG	
MourgosST_MAN	
RajsST_CLERK	

...

20 rows selected.

# 字符串

- 字符串可以是 **SELECT** 列表中的一个字符,数字,日期。
- 日期和字符只能在单引号中出现。
- 每当返回一行时, 字符串被输出一次。

# 字符串

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

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

■ ■ ■

20 rows selected.

# 重复行

默认情况下，查询会返回全部行，包括重复行。

```
SELECT department_id  
FROM   employees;
```

DEPARTMENT_ID	
	90
	90
	90
	60
	60
	60
	50
	50
	50

...

20 rows selected.

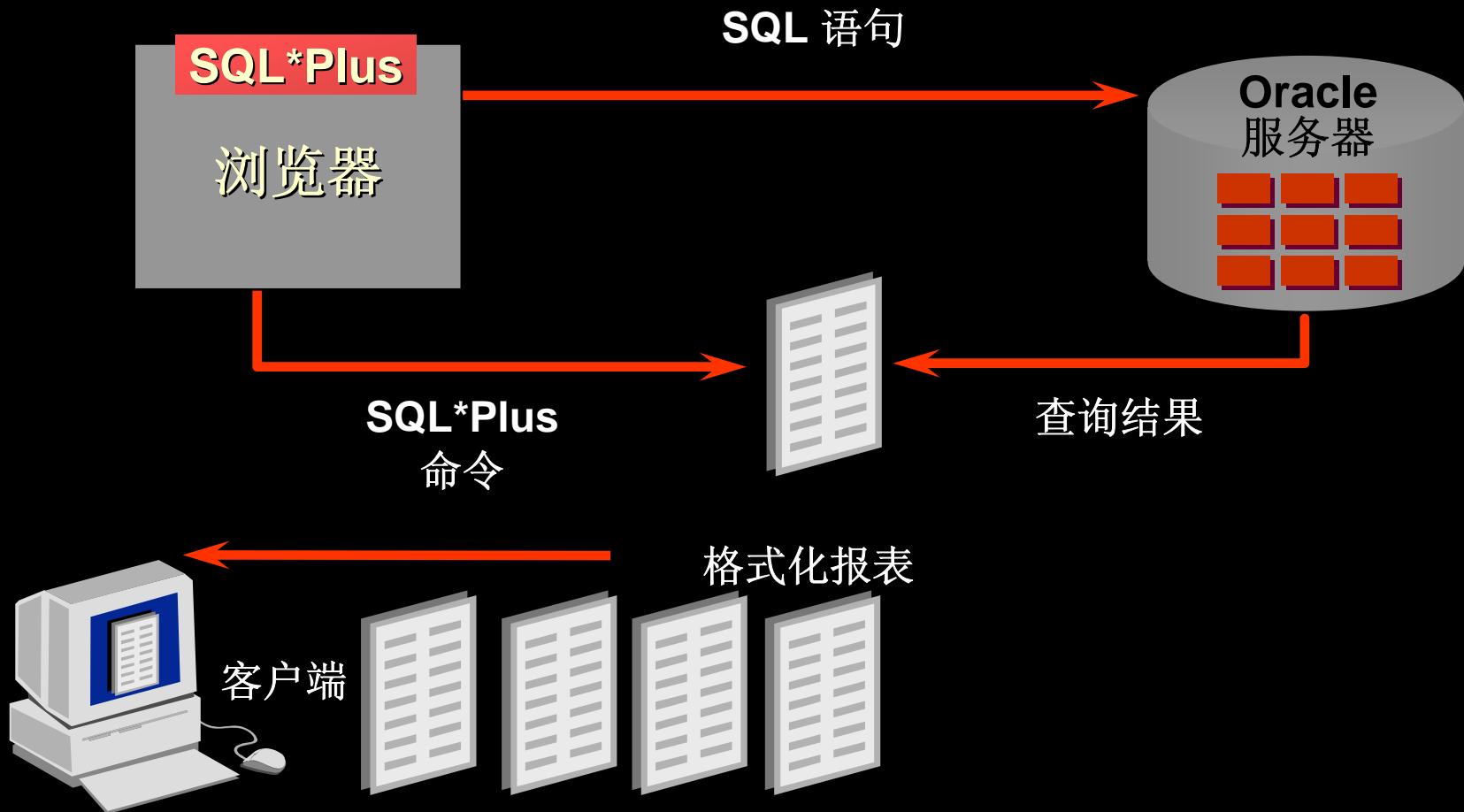
# 删除重复行

在 **SELECT** 子句中使用关键字 '**DISTINCT**' 删除重复行。

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

DEPARTMENT_ID	
	10
	20
	50
	60
	80
	90
	110
8 rows selected.	

# SQL 和 SQL\*Plus





# SQL 语句与 SQL\*Plus 命令

## SQL

- 一种语言
- **ANSI** 标准
- 关键字不能缩写
- 使用语句控制数据库中的表的定义信息和表中的数据

**SQL**  
**statements**

## SQL\*Plus

- 一种环境
- **Oracle** 的特性之一
- 关键字可以缩写
- 命令不能改变数据库中的数据的值
- 集中运行

**SQL\*Plus**  
**commands**

# SQL\*Plus

使用**SQL\*Plus**可以:

- 描述表结构。
- 编辑 **SQL** 语句。
- 执行 **SQL**语句。
- 将 **SQL** 保存在文件中并将**SQL**语句执行结果保存在文件中。
- 在保存的文件中执行语句。
- 将文本文件装入 **SQL\*Plus**编辑窗口。

# 显示表结构

使用 **DESCRIBE** 命令，表示表结构

```
DESC[RIBE] tablename
```

# 显示表结构

```
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)

# 总结

通过本课，您应该可以完成：

- 书写 **SELECT** 语句：
  - 返回表中的全部数据。
  - 返回表中指定列的数据。
  - 使用别名。
- 使用 **SQL\*Plus** 环境，书写，保存和执行 **SQL** 语句和 **SQL\*Plus** 命令。

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

# 2

## 过滤和排序数据

# 目标

通过本章学习，您将可以：

- 在查询中过滤行。
- 在查询中对行进行排序。

# 在查询中过滤行


## EMPLOYEES

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

...

20 rows selected.

返回在 **90** 好部门工作的所有员工的信息



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



# 过滤

- 使用**WHERE** 子句，将不满足条件的行过滤掉。

```
SELECT    * | { [DISTINCT] column/expression [alias], ... }  
FROM      table  
[WHERE    condition(s)];
```

- **WHERE** 子句紧随 **FROM** 子句。

# WHERE 子句

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees
WHERE  department_id = 90 ;
```

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

# 字符和日期

- 字符和日期要包含在单引号中。
- 字符大小写敏感，日期格式敏感。
- 默认的时间格式是 **DD-MON-RR**。

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

# 比较运算

操作符	含义
=	等于
>	大于
>=	大于、等于
<	小于
<=	小于、等于
<>	不等于

# 比较运算

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

LAST_NAME	SALARY
Matos	2600
Vargas	2500

# 其它比较运算

操作符	含义
<b>BETWEEN</b> <b>...AND...</b>	在两个值之间 (包含边界)
<b>IN(set)</b>	等于值列表中的一个
<b>LIKE</b>	模糊查询
<b>IS NULL</b>	空值

# BETWEEN

使用 **BETWEEN** 运算来显示在一个区间内的值。

```
SELECT last_name, salary
FROM   employees
WHERE  salary BETWEEN 2500 AND 3500;
```

↑  
Lower limit

↑  
Upper limit

LAST_NAME	SALARY
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500

# IN

使用 IN 运算显示列表中的值。

```
SELECT employee_id, last_name, salary, manager_id
FROM   employees
WHERE  manager_id IN (100, 101, 201);
```

EMPLOYEE_ID	LAST_NAME	SALARY	MANAGER_ID
202	Fay	6000	201
200	Whalen	4400	101
205	Higgins	12000	101
101	Kochhar	17000	100
102	De Haan	17000	100
124	Mourgos	5800	100
149	Zlotkey	10500	100
201	Hartstein	13000	100

8 rows selected.



# LIKE

- 使用 **LIKE** 运算选择类似的值
- 选择条件可以包含字符或数字：
  - % 代表一个或多个字符。
  - \_ 代表一个字符。

```
SELECT    first_name
FROM      employees
WHERE     first_name LIKE 'S%';
```

# LIKE

- ‘%’和‘\_’可以同时使用。

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

LAST_NAME
Kochhar
Lorentz
Mourgos

- 可以使用 **ESCAPE** 标识符 选择‘%’和‘\_’ 符号。

# NULL

使用 NULL 判断空值。

```
SELECT last_name, manager_id  
FROM   employees  
WHERE  manager_id IS NULL;
```

LAST_NAME	MANAGER_ID
King	

# 逻辑运算

操作符	含义
<b>AND</b>	逻辑并
<b>OR</b>	逻辑或
<b>NOT</b>	逻辑否

# AND

AND 要求的关系为真。

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary >=10000
AND    job_id LIKE '%MAN%';
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
149	Zlotkey	SA_MAN	10500
201	Hartstein	MK_MAN	13000

# OR

OR 要求或关系为真。

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary >= 10000
OR     job_id LIKE '%MAN%';
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
100	King	AD_PRES	24000
101	Kochhar	AD_VP	17000
102	De Haan	AD_VP	17000
124	Mourgos	ST_MAN	5800
149	Zlotkey	SA_MAN	10500
174	Abel	SA_REP	11000
201	Hartstein	MK_MAN	13000
205	Higgins	AC_MGR	12000

8 rows selected.

# NOT

```
SELECT last_name, job_id
FROM   employees
WHERE  job_id
       NOT IN ( 'IT_PROG', 'ST_CLERK', 'SA_REP' );
```

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

10 rows selected.

# 优先级

优先级	
1	算术运算符
2	连接符
3	比较符
4	IS [NOT] NULL, LIKE, [NOT] IN
5	[NOT] BETWEEN
6	NOT
7	AND
8	OR

可以使用括号改变优先级顺序



# 优先级

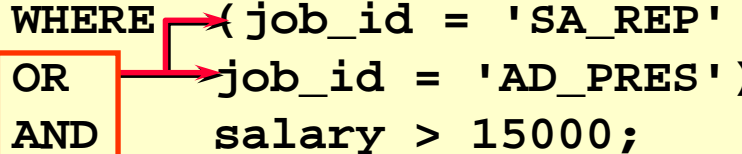
```
SELECT last_name, job_id, salary
FROM   employees
WHERE  job_id = 'SA_REP'
OR     job_id = 'AD_PRES'
AND    salary > 15000;
```

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

# 优先级

使用括号控制执行顺序。

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



LAST_NAME	JOB_ID	SALARY
King	AD_PRES	24000

# ORDER BY子句

- 使用 ORDER BY 子句排序
  - ASC: 升序
  - DESC: 降序
- ORDER BY 子句在SELECT语句的结尾。

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

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

...

20 rows selected.

# 降序排序

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

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

...

20 rows selected.

## 按别名排序

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

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

■ ■ ■

20 rows selected.

# 多个列排序

- 按照ORDER BY 列表的顺序排序。

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

LAST_NAME	DEPARTMENT_ID	SALARY
Whalen	10	4400
Hartstein	20	13000
Fay	20	6000
Mourgos	50	5800
Rajs	50	3500
Davies	50	3100
Matos	50	2600
Vargas	50	2500

...

20 rows selected.

- 可以使用不在SELECT 列表中的列排序。

# 总结

通过本课，您应该可以完成：

- 使用 **WHERE** 子句过滤数据
  - 使用比较运算
  - 使用 **BETWEEN**, **IN**, **LIKE**和 **NULL**运算
  - 使用逻辑运算符 **AND**, **OR**和**NOT**
- 使用 **ORDER BY** 子句进行排序。

```
SELECT      * | { [DISTINCT] column/expression [alias], ... }  
FROM        table  
[WHERE      condition(s)]  
[ORDER BY   { column, expr, alias } [ASC|DESC]];
```





# 3

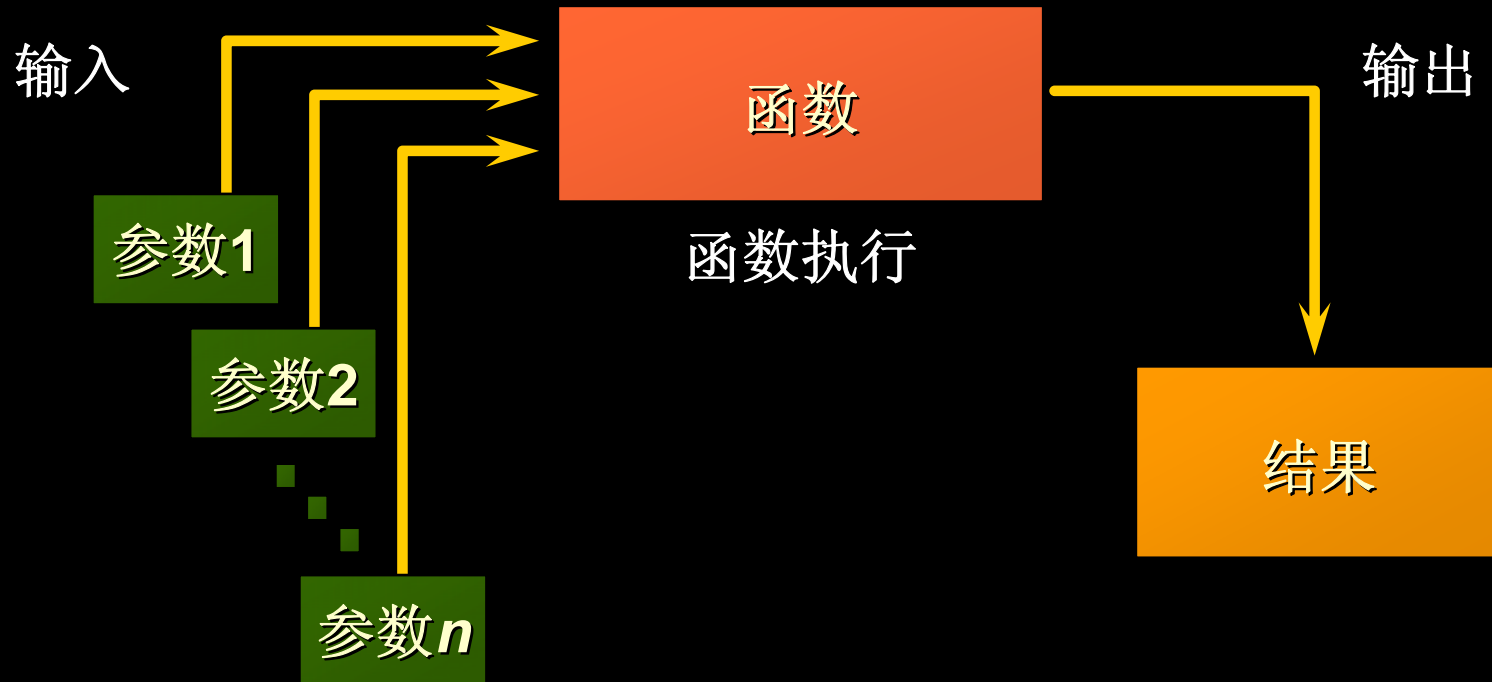
## 单行函数

# 目标

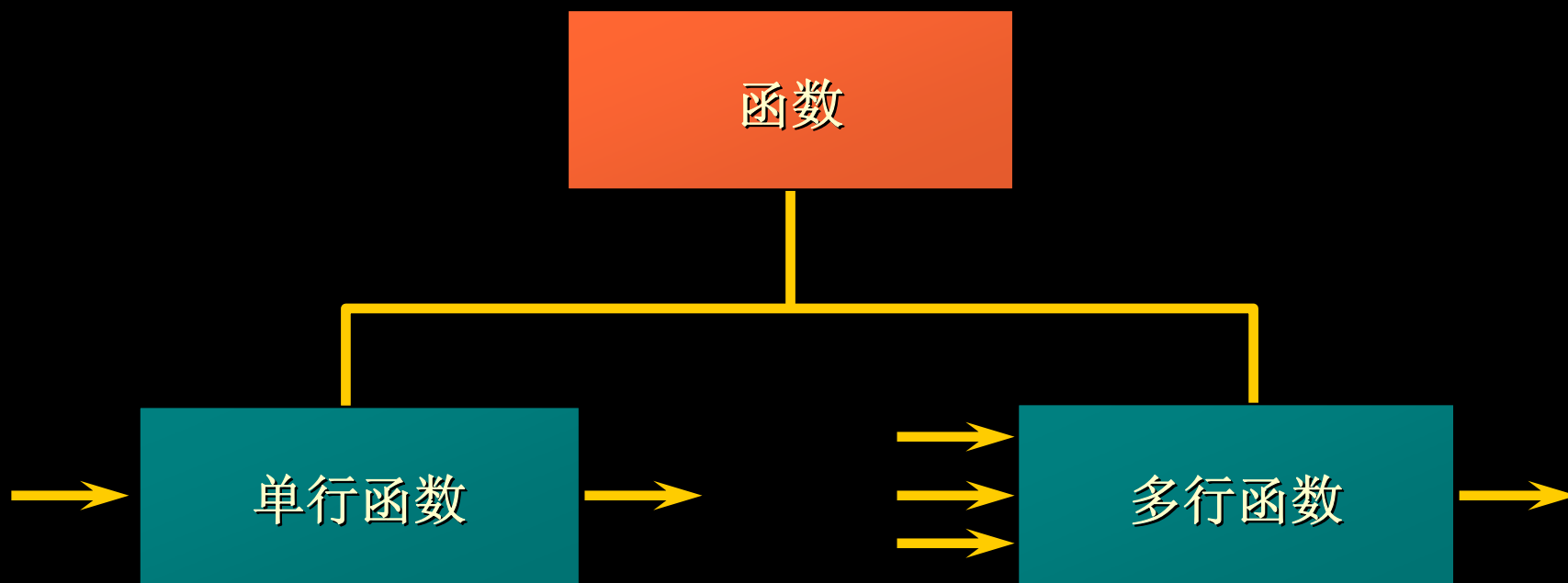
通过本章学习，您将可以：

- **SQL**中不同类型的函数。
- 在 **SELECT** 语句中使用字符，数字和日期函数。
- 描述转换型函数的用途。

# SQL 函数



# 两种 SQL 函数



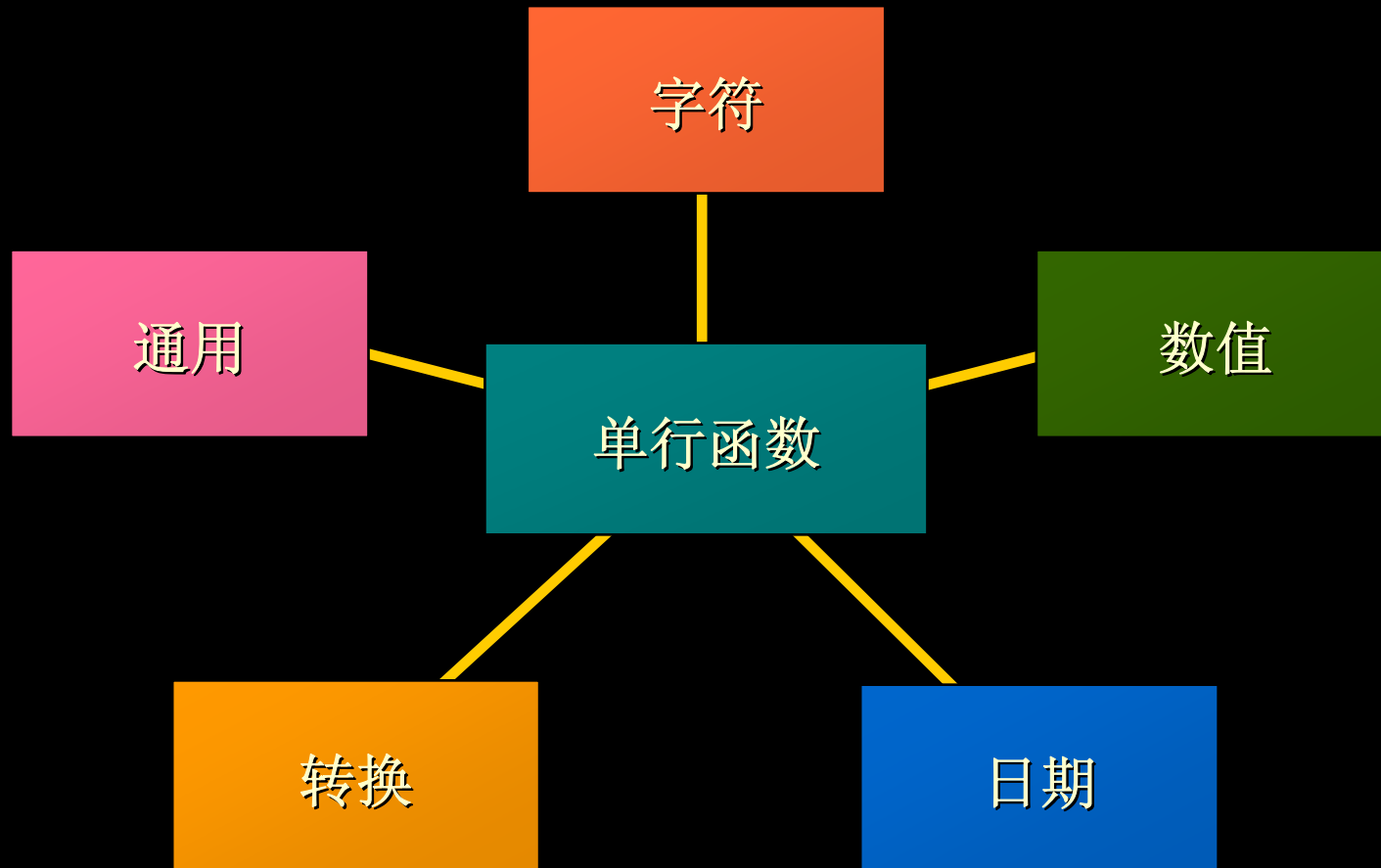
# 单行函数

单行函数:

- 操作数句对象
- 接受函数返回一个结果
- 只对一行进行变换
- 每行返回一个结果
- 可以转换数据类型
- 可以嵌套
- 参数可以是一列或一个值

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

# 单行函数



# 字符函数

## 字符函数

```
graph TD; A[字符函数] --> B[大小写控制函数]; A --> C[字符控制函数]; B --> B1[LOWER]; B --> B2[UPPER]; B --> B3[INITCAP]; C --> C1[CONCAT]; C --> C2[SUBSTR]; C --> C3[LENGTH]; C --> C4[INSTR]; C --> C5[LPAD | RPAD]; C --> C6[TRIM]; C --> C7[REPLACE];
```

### 大小写控制函数

LOWER

UPPER

INITCAP

### 字符控制函数

CONCAT

SUBSTR

LENGTH

INSTR

LPAD | RPAD

TRIM

REPLACE

# 大小写控制函数

这类函数改变字符的大小写。

函数	结果
<code>LOWER( 'SQL Course' )</code>	<code>sql course</code>
<code>UPPER( 'SQL Course' )</code>	<code>SQL COURSE</code>
<code>INITCAP( 'SQL Course' )</code>	<code>Sql Course</code>



# 大小写控制函数

显示员工 **Higgins** 的信息:

```
SELECT employee_id, last_name, department_id
FROM   employees
WHERE  last_name = 'higgins';
no rows selected
```

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

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
205	Higgins	110

# 字符控制函数

这类函数控制字符:

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

# 字符控制函数

```
SELECT employee_id, CONCAT(first_name, last_name) NAME,  
       job_id, LENGTH (last_name),  
       INSTR(last_name, 'a') "Contains 'a'?"  
FROM   employees  
WHERE  SUBSTR(job_id, 4) = 'REP';
```

EMPLOYEE_ID	NAME	JOB_ID	LENGTH(LAST_NAME)	Contains 'a'?
174	EllenAbel	SA_REP	4	0
176	JonathonTaylor	SA_REP	6	2
178	KimberelyGrant	SA_REP	5	3
202	PatFay	MK_REP	3	2

1

2

3

1

2

3

# 数字函数


- **ROUND:** 四舍五入

**ROUND(45.926, 2)**  **45.93**

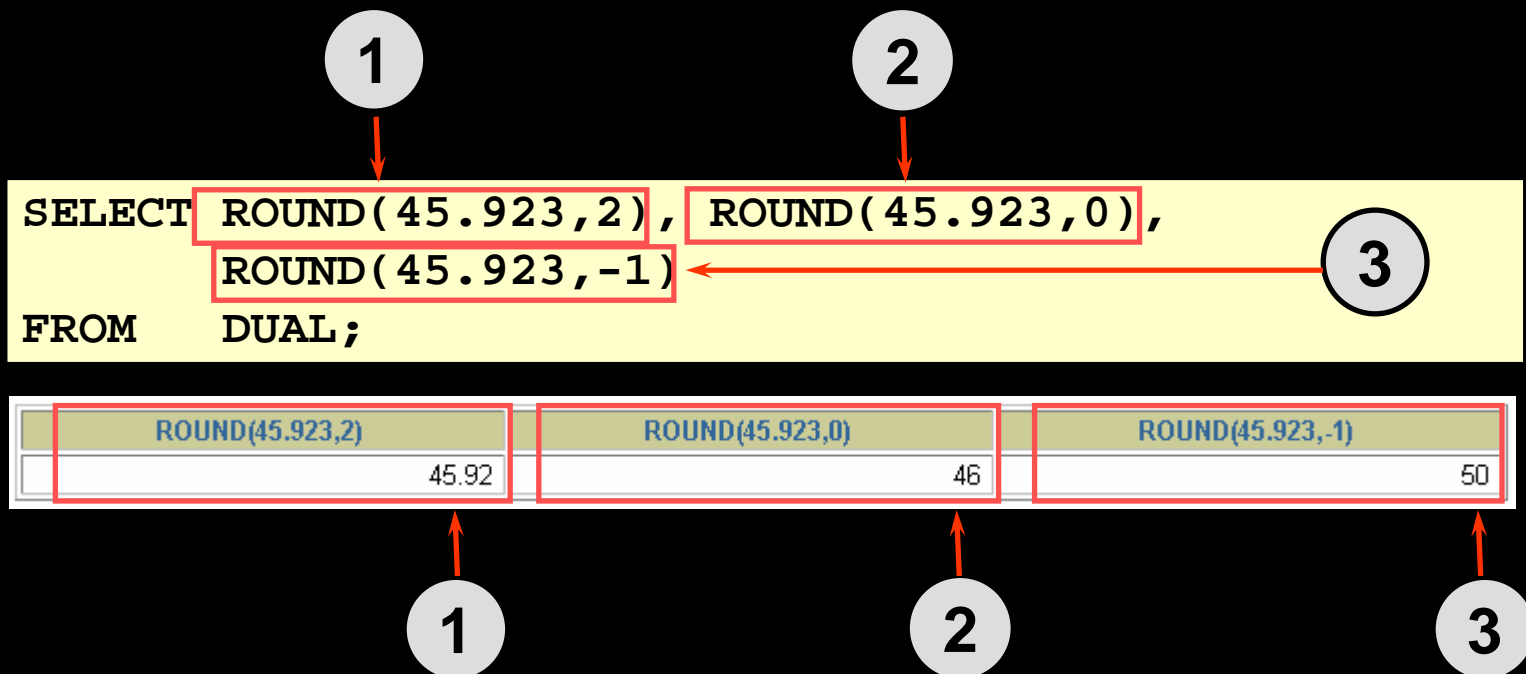
- **TRUNC:** 截断

**TRUNC(45.926, 2)**  **45.92**

- **MOD:** 求余

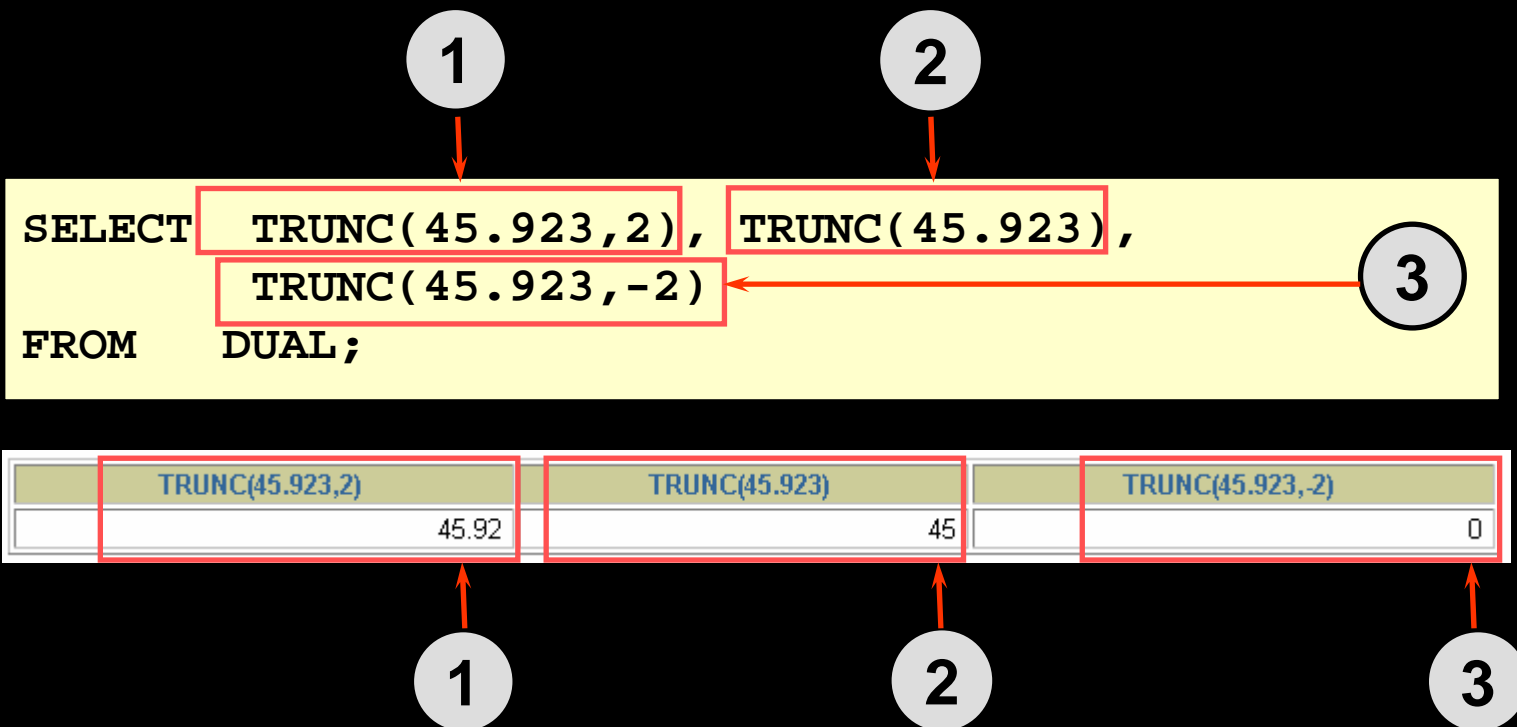
**MOD(1600, 300)**  **100**

# ROUND 函数



DUAL 是一个‘伪表’，可以用来测试函数和表达式。

# TRUNC 函数



# MOD 函数

```
SELECT last_name, salary, MOD(salary, 5000)
FROM   employees
WHERE  job_id = 'SA_REP';
```

LAST_NAME	SALARY	MOD(SALARY,5000)
Abel	11000	1000
Taylor	8600	3600
Grant	7000	2000

# 日期

- **Oracle** 内部使用数字存储日期: 世纪,年,月,日,小时,分钟,秒。
- 默认的时间格式是 **DD-MON-RR**.
  - 可以只指定年的后两位在**20**世纪存放**21**世纪的日期。
  - 同样可以在**21**世纪存放**20**世纪的日期。

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

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



# 日期

函数**SYSDATE** 返回:

- 日期
- 时间

# 日期的数学运算

- 在日期上加上或减去一个数字结果仍为日期。
- 两个日期相减返回日期之间相差的天数。
- 可以用数字除**24**来向日期中加上或减去小时。

# 日期的数学运算

```
SELECT last_name, (SYSDATE-hire_date)/7 AS WEEKS  
FROM employees  
WHERE department_id = 90;
```

LAST_NAME	WEEKS
King	744.245395
Kochhar	626.102538
De Haan	453.245395

# 日期函数

函数	描述
<b>MONTHS_BETWEEN</b>	两个日期相差的月数
<b>ADD_MONTHS</b>	向指定日期中加上若干月数
<b>NEXT_DAY</b>	指定日期的下一个日期
<b>LAST_DAY</b>	本月的最后一天
<b>ROUND</b>	日期四舍五入
<b>TRUNC</b>	日期截断

# 日期函数

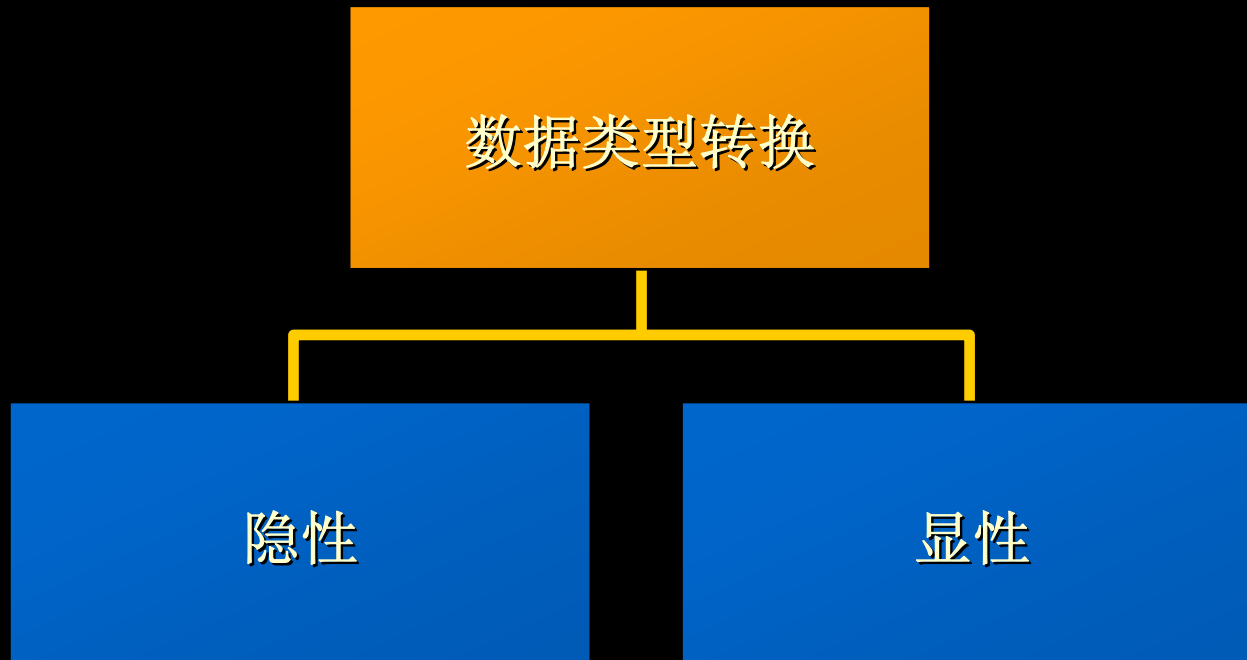
- `MONTHS_BETWEEN ('01-SEP-95', '11-JAN-94')`  
→ 19.6774194
- `ADD_MONTHS ('11-JAN-94', 6)` → '11-JUL-94'
- `NEXT_DAY ('01-SEP-95', 'FRIDAY')`  
→ '08-SEP-95'
- `LAST_DAY('01-FEB-95')` → '28-FEB-95'

# 日期函数

**Assume SYSDATE = '25-JUL-95':**

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

# 转换函数



# 隐式数据类型转换

**Oracle** 自动完成下列转换：

源数据类型	目标数据类型
<b>VARCHAR2 or CHAR</b>	<b>NUMBER</b>
<b>VARCHAR2 or CHAR</b>	<b>DATE</b>
<b>NUMBER</b>	<b>VARCHAR2</b>
<b>DATE</b>	<b>VARCHAR2</b>



# 隐式数据类型转换

表达式计算中, **Oracle** 自动完成下列转换:

源数据类型	目标数据类型
<b>VARCHAR2 or CHAR</b>	<b>NUMBER</b>
<b>VARCHAR2 or CHAR</b>	<b>DATE</b>

# TO\_CHAR 函数对日期的转换

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

格式:

- 必须包含在单引号中而且大小写敏感。
- 可以包含任意的有效的日期格式。
- 可以使用 *fm* 去掉多余的空格或者前导零。
- 与日期指用逗号隔开。

# 日期格式的元素

<b>YYYY</b>	<b>2004</b>
<b>YEAR</b>	<b>TWO THOUSAND AND FOUR</b>
<b>MM</b>	<b>02</b>
<b>MONTH</b>	<b>JULY</b>
<b>MON</b>	<b>JUL</b>
<b>DY</b>	<b>MON</b>
<b>DAY</b>	<b>MONDAY</b>
<b>DD</b>	<b>02</b>

# 日期格式的元素

- 时间格式

HH24:MI:SS AM	15:45:32 PM
---------------	-------------

- 使用双引号向日期中添加字符

DD "of" MONTH	12 of OCTOBER
---------------	---------------

- 日期在月份中的位置

ddspth	fourteenth
--------	------------

# TO\_CHAR 函数对日期的转换

```
SELECT last_name,  
       TO_CHAR(hire_date, 'fmDD Month YYYY')  
       AS HIREDATE  
FROM   employees;
```

LAST_NAME	HIREDATE
King	17 June 1987
Kochhar	21 September 1989
De Haan	13 January 1993
Hunold	3 January 1990
Ernst	21 May 1991
Lorentz	7 February 1999
Mourgos	16 November 1999

...

20 rows selected.

# TO\_CHAR 函数对数字的转换

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

下面是在TO\_CHAR 函数中经常使用的几种格式:

9	数字
0	零
\$	美元符
L	本地货币符号
.	小数点
,	千位符

# TO\_CHAR函数对数字的转换

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

SALARY
\$6,000.00

# TO\_NUMBER 和 TO\_DATE 函数

- 使用 TO\_NUMBER 函数将字符转换成数字:

- `TO_NUMBER(char[, 'format_model'])`

- `TO_DATE(char[, 'format_model'])`



# TO\_NUMBER 和 TO\_DATE 函数

- 使用 TO\_NUMBER 函数将字符转换成数字:

- `TO_NUMBER(char[, 'format_model'])`

- `TO_DATE(char[, 'format_model'])`

# RR 日期格式

当前年	日期	RR 格式	YY 格式
1995	27-OCT-95	1995	1995
1995	27-OCT-17	2017	1917
2001	27-OCT-17	2017	2017
2001	27-OCT-95	1995	2095

		指定的年份:	
		0-49	50-99
当前的年份:	0-49	The return date is in the current century	The return date is in the century before the current one
	50-99	The return date is in the century after the current one	The return date is in the current century

## RR 日期格式

使用RR日期格式查找雇佣日期在1990年之前的员工，  
在1999或现在使用下面的命令会产生相同的结果：

```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-YYYY')  
FROM employees  
WHERE hire_date < TO_DATE('01-Jan-90', 'DD-Mon-RR');
```

LAST_NAME	TO_CHAR(HIR
King	17-Jun-1987
Kochhar	21-Sep-1989
Whalen	17-Sep-1987

# 嵌套函数

- 单行函数可以嵌套。
- 嵌套函数的执行顺序是由内到外。

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



# 嵌套函数

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

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

# 通用函数

这些函数适用于任何数据类型，同时也适用于空值：

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

# NVL 函数

将空值转换成一个已知的值：

- 可以使用的数据类型有日期、字符、数字。
- 函数的一般形式：
  - `NVL(commission_pct,0)`
  - `NVL(hire_date,'01-JAN-97')`
  - `NVL(job_id,'No Job Yet')`

# 使用NVL函数

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

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

...

20 rows selected.

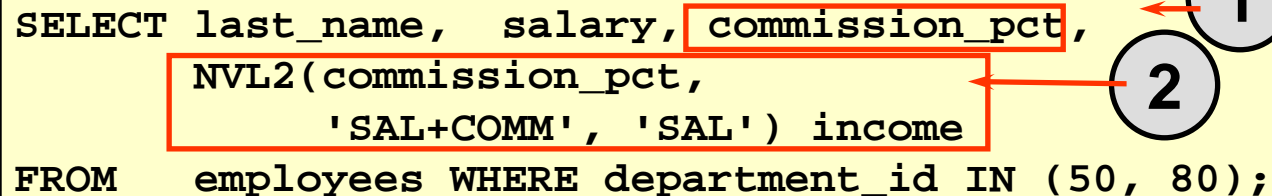
1

2



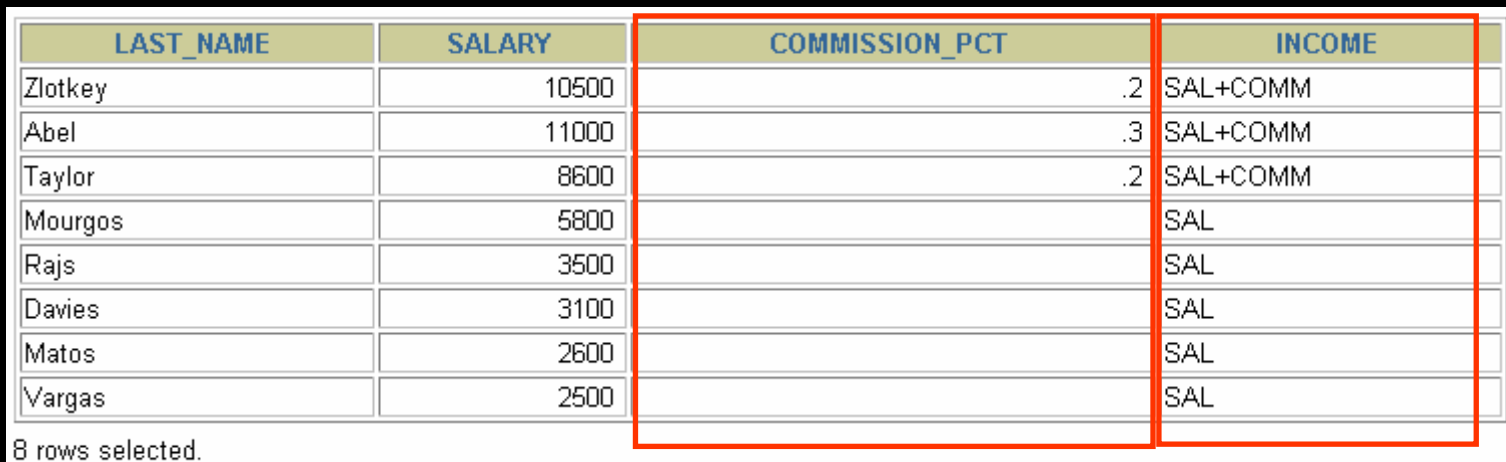
# 使用 NVL2 函数

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



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

8 rows selected.



# 使用 NULLIF 函数

**1**

```
SELECT first_name, LENGTH(first_name) "expr1",  
       last_name,  LENGTH(last_name)  "expr2",  
       NULLIF(LENGTH(first_name), LENGTH(last_name)) result  
FROM   employees;
```

**2**

**3**

FIRST_NAME	expr1	LAST_NAME	expr2	RESULT
Steven	6	King	4	6
Neena	5	Kochhar	7	5
Lex	3	De Haan	7	3
Alexander	9	Hunold	6	9
Bruce	5	Ernst	5	
Diana	5	Lorentz	7	5
Kevin	5	Mourgos	7	5
Trenna	6	Rajs	4	6
Curtis	6	Davies	6	

■ ■ ■

20 rows selected.

**1**

**2**

**3**

# 使用 COALESCE 函数

- COALESCE 与 NVL 相比的优点在于 COALESCE 可以同时处理交替的多个值。
- 如果第一个表达式为空,则返回这个表达式, 对其他的参数进行COALESCE 。

# 使用 COALESCE 函数

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

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

■ ■ ■

20 rows selected.

# 条件表达式

- 在 **SQL** 语句中使用**IF-THEN-ELSE** 逻辑。
- 使用两种方法：
  - **CASE** 表达式
  - **DECODE** 函数

# CASE 表达式

在需要使用 IF-THEN-ELSE 逻辑时:

```
CASE expr WHEN comparison_expr1 THEN return_expr1  
      [WHEN comparison_expr2 THEN return_expr2  
      WHEN comparison_exprn THEN return_exprn  
      ELSE else_expr]  
END
```

# CASE 表达式

下面是使用**case**表达式的一个例子：

```
SELECT last_name, job_id, salary,  
       CASE job_id WHEN 'IT_PROG' THEN 1.10*salary  
                   WHEN 'ST_CLERK' THEN 1.15*salary  
                   WHEN 'SA_REP' THEN 1.20*salary  
       ELSE salary END "REVISED_SALARY"  
FROM employees;
```

LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
...			
Lorentz	IT_PROG	4200	4620
Mourgos	ST_MAN	5800	5800
Rajs	ST_CLERK	3500	4025
...			
Gietz	AC_ACCOUNT	8300	8300

20 rows selected.

# DECODE 函数

在需要使用 IF-THEN-ELSE 逻辑时:

```
DECODE(col/expression, search1, result1  
      [, search2, result2, ..., ]  
      [, default])
```



# DECODE 函数

```
SELECT last_name, job_id, salary,  
       DECODE(job_id, 'IT_PROG', 1.10*salary,  
                 'ST_CLERK', 1.15*salary,  
                 'SA_REP', 1.20*salary,  
                 salary)  
       REVISED_SALARY  
FROM   employees;
```

LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
...			
Lorentz	IT_PROG	4200	4620
Mourgos	ST_MAN	5800	5800
Rajs	ST_CLERK	3500	4025
...			
Gietz	AC_ACCOUNT	8300	8300

20 rows selected.

# DECODE 函数

使用**decode**函数的一个例子：

```
SELECT last_name, salary,  
       DECODE (TRUNC(salary/2000, 0),  
               0, 0.00,  
               1, 0.09,  
               2, 0.20,  
               3, 0.30,  
               4, 0.40,  
               5, 0.42,  
               6, 0.44,  
               0.45) TAX_RATE  
FROM   employees  
WHERE  department_id = 80;
```

# 总结

通过本章学习，您应该学会：

- 使用函数对数据进行计算
- 使用函数修改数据
- 使用函数控制一组数据的输出格式
- 使用函数改变日期的显示格式
- 使用函数改变数据类型
- 使用 **NVL** 函数
- 使用 **IF-THEN-ELSE** 逻辑

# 4

## 多表查询

# 目标

通过本章学习，您将可以：

- 使用等值和不等值连接在**SELECT** 语句中查询多个表中的数据。
- 使用外连接查询不满足连接条件的数据。
- 使用自连接。

# 从多个表中获取数据

**EMPLOYEES**

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

**DEPARTMENTS**

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



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

# 笛卡尔集

- 笛卡尔集会在下面条件下产生：
  - 省略连接条件
  - 连接条件无效
  - 所有表中的所有行互相连接
- 为了避免笛卡尔集，可以在 **WHERE** 加入有效的连接条件。

# 笛卡尔集

**EMPLOYEES (20行)**

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

20 rows selected.

**DEPARTMENTS (8行)**

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

8 rows selected.

笛卡尔集:  
**20x8=160行** →

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

160 rows selected.



# 连接的类型

**Oracle 提供的连接 (8i 或更早):**

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

**适用于SQL: 1999的连接:**

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

# Oracle 连接

使用连接在多个表中查询数据。

```
SELECT    table1.column, table2.column  
FROM      table1, table2  
WHERE     table1.column1 = table2.column2;
```

- 在 **WHERE** 字句中写入连接条件。
- 在表中有相同列时，在列名之前加上表名前缀。

# 等值连接

EMPLOYEES

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

...

↑  
外键

DEPARTMENTS

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

...

↑  
主键

# 等值连接

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

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

■ ■ ■

19 rows selected.

# 多个连接条件与 AND 操作符

**EMPLOYEES**

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

...

**DEPARTMENTS**

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

...

# 区分重复的列名

- 使用表名前缀在多个表中区分相同的列。
- 使用表名可以提高效率。
- 在不同表中具有相同列名的列可以用别名加以区分。

# 表的别名

- 使用别名可以简化查询。
- 使用表名前缀可以提高执行效率。

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

# 连接多个表

EMPLOYEES

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

20 rows selected.

DEPARTMENTS

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

8 rows selected.

LOCATIONS

LOCATION_ID	CITY
1400	Southlake
1500	South San Francisco
1700	Seattle
1800	Toronto
2500	Oxford

- 连接  $n$  个表,至少需要  $n-1$  个连接条件。 例如: 连接三个表, 至少需要两个连接条件。



# 非等值连接

**EMPLOYEES**

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

■ ■ ■

20 rows selected.

**JOB\_GRADES**

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



**EMPLOYEES**表中的列工资  
应在**JOB\_GRADES**表中的最高  
工资与最低工资之间

# 非等值连接

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

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

■ ■ ■

20 rows selected.

# 外连接

## DEPARTMENTS

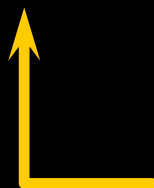
DEPARTMENT_NAME	DEPARTMENT_ID
Administration	10
Marketing	20
Shipping	50
IT	60
Sales	80
Executive	90
Accounting	110
Contracting	190

8 rows selected.

## EMPLOYEES

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

20 rows selected.



190号部门没有员工

# 外连接语法

- 使用外连接可以查询不满足连接条件的数据。
- 外连接的符号是 (+)。

```
SELECT table1.column, table2.column  
FROM   table1, table2  
WHERE  table1.column(+) = table2.column;
```

```
SELECT table1.column, table2.column  
FROM   table1, table2  
WHERE  table1.column = table2.column(+);
```

# 外连接

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

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

20 rows selected.

# 自连接

**EMPLOYEES (WORKER)**

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

...

**EMPLOYEES (MANAGER)**

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

...



**WORKER** 表中的**MANAGER\_ID** 和 **MANAGER** 表中的  
**MANAGER\_ID**相等

# 自连接

```
SELECT worker.last_name || ' works for '
       || manager.last_name
FROM   employees worker, employees manager
WHERE  worker.manager_id = manager.employee_id ;
```

WORKER.LAST_NAME  'WORKSFOR'  MANAGER.LAST_NAME
Kochhar works for King
De Haan works for King
Mourgos works for King
Zlotkey works for King
Hartstein works for King
Whalen works for Kochhar
Higgins works for Kochhar
Hunold works for De Haan
Ernst works for Hunold

19 rows selected.

# 使用SQL: 1999 语法连接

使用连接从多个表中查询数据:

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



# 叉集

- 使用**CROSS JOIN** 子句使连接的表产生叉集。
- 叉集和笛卡尔集是相同的。

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

LAST_NAME	DEPARTMENT_NAME
King	Administration
Kochhar	Administration
De Haan	Administration
Hunold	Administration

160 rows selected.

# 自然连接

- **NATURAL JOIN** 子句，会以两个表中具有相同名字的列为条件创建等值连接。
- 在表中查询满足等值条件的数据。
- 如果只是列名相同而数据类型不同，则会产生错误。

# 自然连接

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

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

8 rows selected.

# 使用 USING 子句创建连接

- 在 **NATURAL JOIN** 子句创建等值连接时，可以使用 **USING** 子句指定等值连接中需要用到的列。
- 使用 **USING** 可以在有多个列满足条件时进行选择。
- 不要给选中的列中加上表名前缀或别名。
- **NATURAL JOIN** 和 **USING** 子句经常同时使用。

# USING 子句

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

EMPLOYEE_ID	LAST_NAME	LOCATION_ID
200	Whalen	1700
201	Hartstein	1800
202	Fay	1800
124	Mourgos	1500
141	Rajs	1500
142	Davies	1500
143	Matos	1500
144	Vargas	1500
103	Hunold	1400

19 rows selected.

# 使用ON 子句创建连接

- 自然连接中是以具有相同名字的列为连接条件的。
- 可以使用 ON 子句指定额外的连接条件。
- 这个连接条件是与其它条件分开的。
- ON 子句使语句具有更高的易读性。

# ON 子句

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

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

...

19 rows selected.

# 使用ON 子句创建多表连接

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

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

...

19 rows selected.



# 内连接 与 外连接

- 在**SQL: 1999**中，内连接只返回满足连接条件的数据。
- 两个表在连接过程中除了返回满足连接条件的行以外还返回左（或右）表中不满足条件的行，这种连接称为左（或右）外联接。
- 两个表在连接过程中除了返回满足连接条件的行以外还返回两个表中不满足条件的行，这种连接称为满 外联接。

# 左外联接

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

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing

...

De Haan	90	Executive
Kochhar	90	Executive
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		

20 rows selected.

## 右外联接

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

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

20 rows selected.

# 满外联接

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

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
...		
De Haan	90	Executive
Kochhar	90	Executive
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		
		Contracting

21 rows selected.

## 增加连接条件

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

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



# 5

## 分组函数

# 目标

通过本章学习，您将可以：

- 了解组函数。
- 描述组函数的用途。
- 使用**GROUP BY** 字句数据分组。
- 使用**HAVING** 字句过滤分组结果集。



# 什么是分组函数

分组函数作用于一组数据，并对一组数据返回一个值。

**EMPLOYEES**

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

...

20 rows selected.

表 **EMPLOYEES**  
中的工资最大值

MAX(SALARY)
24000

# 组函数类型

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

# 组函数语法

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

# AVG（平均值）和 SUM（合计）函数

可以对数值型数据使用AVG 和 SUM 函数。

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

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

# MIN（最小值） 和 MAX（最大值） 函数

可以对任意数据类型的数据使用 MIN 和 MAX 函数。

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

MIN(HIRE_	MAX(HIRE_
17-JUN-87	29-JAN-00

# COUNT（计数）函数

COUNT（\*）返回表中记录总数。

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

COUNT(*)	
	5

# COUNT（计数）函数

- COUNT(*expr*) 返回 *expr*不为空的记录总数。

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

COUNT(COMMISSION\_PCT)

3

# DISTINCT 关键字

- `COUNT(DISTINCT expr)` 返回 *expr* 非空且不重复的记录总数

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

COUNT(DISTINCTDEPARTMENT_ID)
7



# 组函数与空值

组函数忽略空值。

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

AVG(COMMISSION_PCT)
.2125

# 在组函数中使用NVL函数

NVL函数使分组函数无法忽略空值。

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

AVG(NVL(COMMISSION_PCT,0))
.0425

# 分组数据

## EMPLOYEES

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

...

20 rows selected.

4400

9500

3500

6400

10033

求出  
EMPLOYEES  
表中各  
部门的  
平均工资

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

# 分组数据: GROUP BY 子句语法

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

可以使用GROUP BY 子句将表中的数据分成若干组

# GROUP BY 子句

在SELECT 列表中所有未包含在组函数中的列都应该包含在 GROUP BY 子句中。

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

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

8 rows selected.

# GROUP BY 子句

包含在 GROUP BY 子句中的列不必包含在SELECT 列表中。

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

AVG(SALARY)	
	4400
	9500
	3500
	6400
	10033.3333
	19333.3333
	10150
	7000

# 使用多个列分组

## EMPLOYEES

DEPARTMENT_ID	JOB_ID	SALARY
90	AD_PRES	24000
90	AD_VP	17000
90	AD_VP	17000
60	IT_PROG	9000
60	IT_PROG	6000
60	IT_PROG	4200
50	ST_MAN	5800
50	ST_CLERK	3500
50	ST_CLERK	3100
50	ST_CLERK	2600
50	ST_CLERK	2500
80	SA_MAN	10500
80	SA_REP	11000
80	SA_REP	8600
...		
20	MK_REP	6000
110	AC_MGR	12000
110	AC_ACCOUNT	8300

20 rows selected.

使用多个列  
进行分组

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

13 rows selected.

## 在GROUP BY 子句中包含多个列

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

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

13 rows selected.



# 非法使用组函数

所用包含于**SELECT** 列表中，而未包含于组函数中的列都必须包含于 **GROUP BY** 子句中。

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

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

**GROUP BY** 子句中缺少列

# 非法使用组函数

- 不能在 **WHERE** 子句中使用组函数。
- 可以在**HAVING** 子句中使用组函数。

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

```
WHERE     AVG(salary) > 8000
```

```
*
```

```
ERROR at line 3:
```

```
ORA-00934: group function is not allowed here
```

**WHERE** 子句中不能使用组函数

# 过滤分组

## EMPLOYEES

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

20 rows selected.

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

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

# 过滤分组: **HAVING** 子句

使用 **HAVING** 过滤分组:

1. 行已经被分组。
2. 使用了组函数。
3. 满足**HAVING** 子句中条件的分组将被显示。

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

# HAVING 子句

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

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

# HAVING 子句

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

JOB_ID	PAYROLL
IT_PROG	19200
AD_PRES	24000
AD_VP	34000

# 嵌套组函数

显示平均工资的最大值

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

MAX(AVG(SALARY))
19333.3333

# 总结

通过本章学习，您已经学会：

- 使用组函数。
- 在查询中使用 **GROUP BY** 子句。
- 在查询中使用 **HAVING** 子句。

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



# 6

## 子查询

# 目标

通过本章学习，您将可以：

- 描述子查询可以解决的问题
- 定义子查询。
- 列句子查询的类型。
- 书写单行子查询和多行字查询。

# 使用子查询解决问题

谁的工资比 **Abel** 高？

Main Query:



谁的工资比 **Abel** 高？

Subquery



**Abel**的工资是多少？



# 子查询语法

```
SELECT    select_list  
FROM      table  
WHERE     expr operator
```

```
( SELECT      select_list  
  FROM        table );
```

- 子查询 (内查询) 在主查询之前一次执行完成。
- 子查询的结果被主查询使用 (外查询)。

# 子查询

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

LAST_NAME
King
Kochhar
De Haan
Hartstein
Higgins

# 注意事项

- 子查询要包含在括号内。
- 将子查询放在比较条件的右侧。
- 除非进行**Top-N** 分析，否则不要在子查询中使用**ORDER BY** 子句。
- 单行操作符对应单行子查询，多行操作符对应多行子查询。

# 子查询类型

- 单行子查询



- 多行子查询



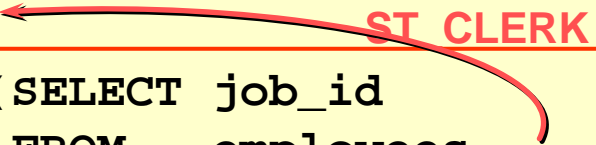
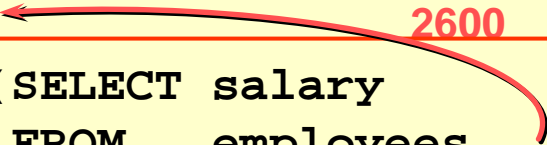
# 单行子查询

- 只返回一行。
- 使用单行比较操作符。

操作符	含义
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to



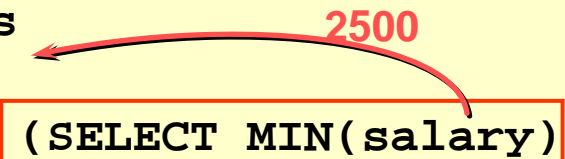
# 执行单行子查询

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

LAST_NAME	JOB_ID	SALARY
Rajs	ST_CLERK	3500
Davies	ST_CLERK	3100

## 在子查询中使用组函数

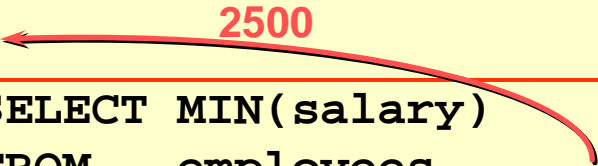
```
SELECT last_name, job_id, salary
FROM   employees
WHERE  salary = (SELECT MIN(salary)
```



LAST_NAME	JOB_ID	SALARY
Vargas	ST_CLERK	2500

# 子查询中的 HAVING 子句

- 首先执行子查询。
- 向主查询中的HAVING 子句返回结果。

```
SELECT    department_id, MIN(salary)
FROM      employees
GROUP BY  department_id
HAVING    MIN(salary) >  2500
      (SELECT MIN(salary)
FROM      employees
WHERE     department_id = 50);
```

# 非法使用子查询

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

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

多行子查询使用单行比较符

# 子查询中的空值问题

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

no rows selected

子查询不返回任何行

# 多行子查询

- 返回多行。
- 使用多行比较操作符。

操作符	含义
<b>IN</b>	等于列表中的任何一个
<b>ANY</b>	和子查询返回的任意一个值比较
<b>ALL</b>	和子查询返回的所有值比较

# 在多行子查询中使用 ANY 操作符

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

9000, 6000, 4200

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

10 rows selected.

# 在多行子查询中使用 ALL 操作符

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

9000, 6000, 4200

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



# 子查询中的空值问题

```
SELECT emp.last_name
FROM   employees emp
WHERE  emp.employee_id NOT IN
                                (SELECT mgr.manager_id
                                FROM   employees mgr);
```

no rows selected

# 总结

通过本章学习，您已经学会：

- 在什么时候遇到什么问题应该使用子查询。
- 在查询是基于未知的值时应使用子查询。

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

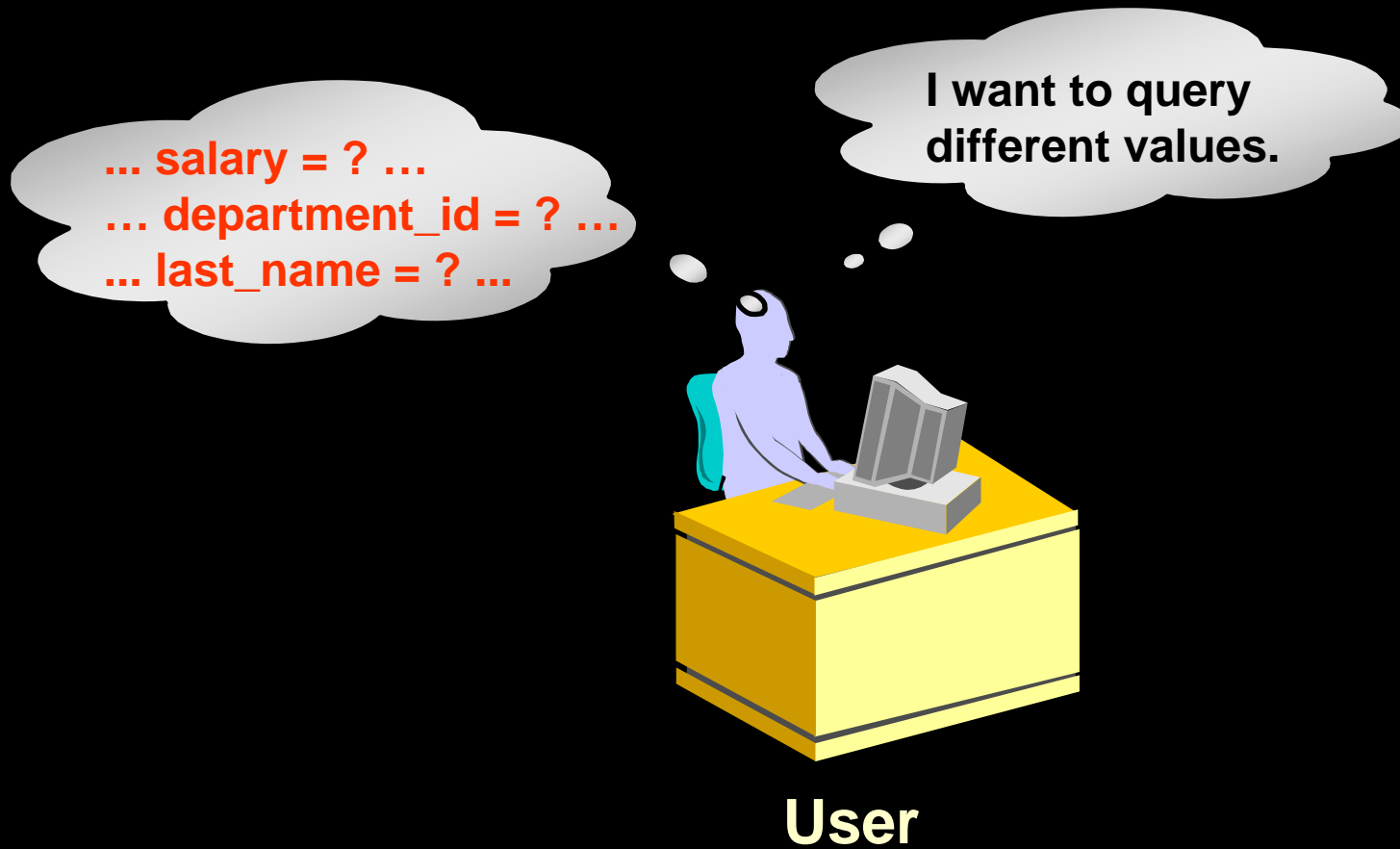


# 目标

通过本章学习，您将可以：

- 在查询中使用变量。
- 熟悉 **iSQL\*Plus** 环境。
- 使输出更便于理解。
- 创建和执行脚本。

# 变量



# 变量

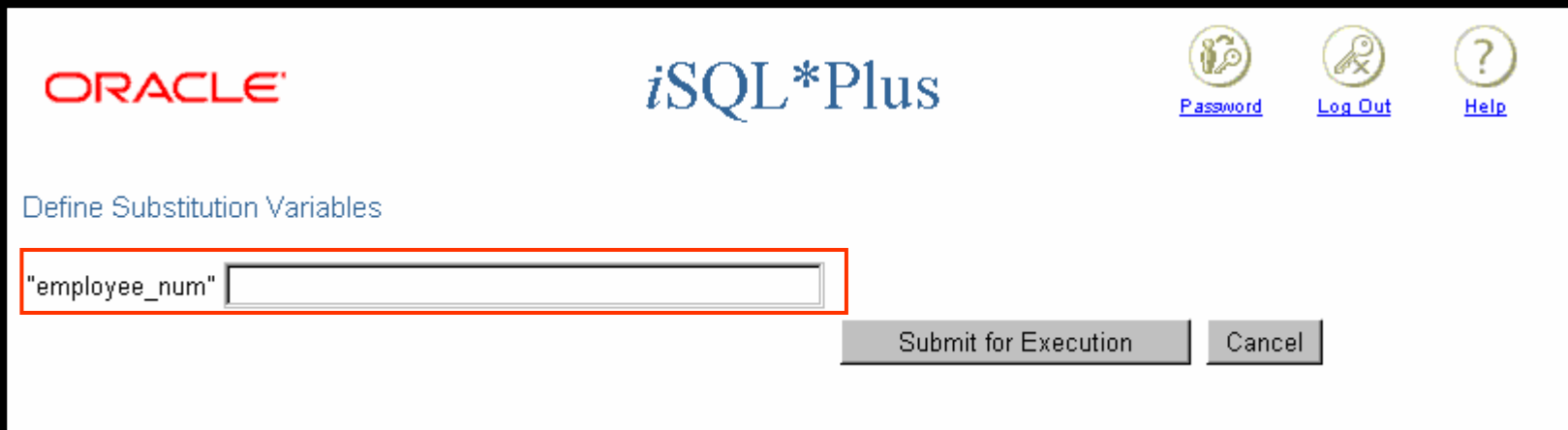
## 使用SQL\*Plus 变量:

- 临时存储值
  - 单个 (&)
  - 两个 (&&)
  - 定义命令
- 在SQL语句中改变变量的值。
- 动态修改开头和结尾。

## & 变量



在变量名前加前缀 (&) 使用户输入值。




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



The screenshot shows the Oracle iSQL\*Plus web interface. At the top, there is the Oracle logo, the text 'iSQL\*Plus', and three icons: a key for 'Password', a key with a slash for 'Log Out', and a question mark for 'Help'. Below this, the text 'Define Substitution Variables' is displayed. A dialog box is open, showing a text input field with the label '"employee\_num"'. The input field is empty. To the right of the input field are two buttons: 'Submit for Execution' and 'Cancel'.

# & 变量



[Password](#)[Log Out](#)[Help](#)

Define Substitution Variables

"employee\_num"  1

2

old 3: WHERE employee\_id = &employee\_num  
new 3: WHERE employee\_id = 101

EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID
101	Kochhar	17000	90



# 字符和日期型变量

在子符和日期两端加单引号。

```
SELECT last_name, department_id, salary*12
FROM employees
WHERE job_id = '&job_title' ;
```

Define Substitution Variables

"job\_title"

Submit for Execution

Cancel

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

# 指定列名、表达式和文本

使用变量可以提供下面的内容:

- **WHERE** 条件
- **ORDER BY** 子句
- 列表表达式
- 表名
- 整个 **SELECT** 语句

# 指定列名、表达式和文本

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

## Define Substitution Variables

"column\_name" salary  
"condition" salary > 15000  
"order\_column" last\_name

Submit for Execution

Cancel

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

# 定义变量

- 可以使用**DEFINE** 命令提前定义**iSQL\*Plus** 变量。

**DEFINE variable = value** 创建一个字符型用户变量

- 使用 **DEFINE** 定义的变量名字中包含空格时，变量名应包含在单引号中。
- 定义的边令在会话级有效。

# DEFINE 和 UNDEFINE 命令

- 定义命令在下列条件下失效:
  - UNDEFINE 命令
  - 退出 *iSQL\*Plus*
- 可以重复使用 DEFINE 命令改变变量。

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

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

# DEFINE 命令与& 变量

- 使用 **DEFINE** 创建变量。

```
DEFINE employee_num = 200
```

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

EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID
200	Whalen	4400	10

# && 变量

使用 (&&) 避免为同一变量重复赋值。

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

Define Substitution Variables

"column\_name"

Submit for Execution

Cancel

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

...

20 rows selected.

# VERIFY 命令

使用 **VERIFY** 在 **iSQL\*Plus** 中显示变量被替代前和变量被替代后的**SQL**语句。

```
SET VERIFY ON
```

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

"employee\_num"

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



# iSQL\*Plus 环境

- 使用 **SET** 命令控制当前会话。

```
SET system_variable value
```

- 使用 **SHOW** 命令显示当前的设置。

```
SET ECHO ON
```

```
SHOW ECHO  
echo ON
```

# SET 命令

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

SET HEADING OFF

SHOW HEADING

**HEADING OFF**

# iSQL\*Plus 格式命令

- COLUMN [*column option*]
- TTITLE [*text* | OFF | ON]
- BTITLE [*text* | OFF | ON]
- BREAK [ON *report\_element*]

# COLUMN 命令

控制列的输出:

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

- **CLE[AR]:** 清除列格式
- **HEA[DING] *text*:** 设置列头
- **FOR[MAT] *format*:** 改变列的输出格式
- **NOPRINT | PRINT**
- **NULL**

# COLUMN 命令

- 创建列头:

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

- 显示 LAST\_NAME 列的当前格式。

```
COLUMN last_name
```

- 清除 LAST\_NAME 列的当前格式设置

```
COLUMN last_name CLEAR
```

## COLUMN 格式

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

# BREAK 命令

使用 **BREAK** 命令去重。

```
BREAK ON job_id
```

# TTITLE 和 BTITLE 命令

- 显示报告头和报告尾

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

- 设置报告头。

```
TTITLE 'Salary|Report'
```

- 摄制报告尾。

```
BTITLE 'Confidential'
```



# TTITLE 和 BTITLE 命令

- 显示报告头和报告尾。

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

- 设置报告头。

```
TTITLE 'Salary|Report'
```

- 设置报告尾。

```
BTITLE 'Confidential'
```

# 使用脚本创建报告

1. 书写并测试 **SQL SELECT** 语句。
2. 保存 **SELECT** 语句到脚本文件。
3. 在编辑器中执行脚本。
4. 在 **SELECT** 语句前添加格式命令。
5. 在 **SELECT** 语句后添加终止符。

# 使用脚本创建报告

6. 在 **SELECT** 后清除格式设置。
7. 保存脚本。
8. 在 **iSQL\*Plus** 的文本框中加载脚本, 点击执行按钮运行脚本。

# 报告

Fri Sep 28

Employee  
Report

page 1

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

Confidential

■ ■ ■

ORACLE

# 报告

Fri Sep 28

Employee  
Report

page 1

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

Confidential

■ ■ ■

# 总结

通过本章学习，您已经学会：

- 使用 **iSQL\*Plus** 变量临时存储值。
- 使用 **SET** 命令控制当前 **iSQL\*Plus** 环境。
- 使用 **COLUMN** 命令控制列的输出。
- 使用 **BREAK** 命令去重并将结果积分组。
- 使用 **TTITLE** 和 **BTITLE** 显示报告头和报告尾。

# Practice 7 Overview

**This practice covers the following topics:**

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

# 8

## 处理数据



# 目标

通过本章学习，您将可以：

- 使用 **DML** 语句
- 向表中插入数据
- 更新表中数据
- 从表中删除数据
- 将表中数据和并
- 控制事务

# 数据控制语言

- **DML** 可以在下列条件下执行：
  - 向表中插入数据
  - 修改现存数据
  - 删除现存数据
- 事务是由完成若干项工作的**DML**语句组成的。

# 插入数据

新行

## 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

向 DEPARTMENTS  
表中插入  
新的记录



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

ORACLE

# INSERT 语句语法

- 使用 **INSERT** 语句向表中插入数据。

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

- 使用这种语法一次只能向表中插入一条数据。

# 插入数据

- 为每一列添加一个新值。
- 按列的默认顺序列出各个列的值。
- 在 **INSERT** 子句中随意列出列名和他们的值。
- 字符和日期型数据应包含在单引号中。

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

# 向表中插入空值

- 隐式方式: 在列名表中省略该列的值。

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

- 显示方式: 在VALUES 子句中指定空值。

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

# 插入指定的值

**SYSDATE** 记录当前系统的日期和时间。

```
INSERT INTO employees (employee_id,  
                        first_name, last_name,  
                        email, phone_number,  
                        hire_date, job_id, salary,  
                        commission_pct, manager_id,  
                        department_id)  
VALUES (113,  
        'Louis', 'Popp',  
        'LPOPP', '515.124.4567',  
        SYSDATE, 'AC_ACCOUNT', 6900,  
        NULL, 205, 100);
```

1 row created.

# 插入指定的值

- 加入新员工

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

1 row created.

- 检查插入的数据

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



# 创建脚本

- 在**SQL** 语句中使用**&** 变量指定列值。
- **&** 变量放在**VALUES**子句中。

```
INSERT INTO departments
      (department_id, department_name, location_id)
VALUES  (&department_id, '&department_name', &location);
```

## Define Substitution Variables

"department\_id"

"department\_name"

"location"

Submit for Execution

Cancel

**1 row created.**

# 从其它表中拷贝数据

- 在 **INSERT** 语句中加入子查询。

```
INSERT INTO sales_reps(id, name, salary, commission_pct)
  SELECT employee_id, last_name, salary, commission_pct
 FROM    employees
 WHERE   job_id LIKE '%REP%';
```

4 rows created.


- 不必书写 **VALUES** 子句。
- 子查询中的值列表应于 **INSERT** 子句中的列名对应。

# 更新数据

## EMPLOYEES

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

## 更新 EMPLOYEES 表



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

# UPDATE 语句语法

- 使用 **UPDATE** 语句更新数据。

```
UPDATE      table  
SET         column = value [, column = value, ...]  
[WHERE      condition];
```

- 可以一次更新多条数据。

# 更新数据

- 使用 **WHERE** 子句指定需要更新的数据。

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

- 如果省略**WHERE**子句，则表中的所有数据都将被更新。

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

# 在UPDATE语句中使用子查询

更新 114号员工的工作和工资使其与 205号员工相同。

```
UPDATE    employees
SET       job_id   = (SELECT  job_id
                      FROM    employees
                      WHERE    employee_id = 205),
          salary   = (SELECT  salary
                      FROM    employees
                      WHERE    employee_id = 205)
WHERE     employee_id = 114;
1 row updated.
```

# 在UPDATE语句中使用子查询

在 UPDATE 中使用子查询，使更新基于另一个表中的数据。

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

1 row updated.
```

# 更新中的数据完整性错误

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

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

不存在 **55** 号部门



# 删除数据

## DEPARTMENTS

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

从表DEPARTMENTS 中删除一条记录。

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

# DELETE 语句

使用 **DELETE** 语句从表中删除数据。

```
DELETE [FROM]    table  
[WHERE           condition];
```

# 删除数据

- 使用WHERE 子句指定删除的记录。

```
DELETE FROM departments
WHERE department_name = 'Finance';
1 row deleted.
```

- 如果省略WHERE子句，则表中的全部数据将被删除。

```
DELETE FROM copy_emp;
22 rows deleted.
```

# 在 DELETE 中使用子查询

在 DELETE 中使用子查询，使删除基于另一个表中的数据。

```
DELETE FROM employees
WHERE department_id =
    (SELECT department_id
     FROM departments
     WHERE department_name LIKE '%Public%');

1 row deleted.
```

# 删除中的数据完整性错误

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

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

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

# 在INSERT语句中使用子查询

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

1 row created.

# 在INSERT语句中使用子查询

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

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

6 rows selected.

# 在DML语句中使用 WITH CHECK OPTION 关键字

- 使用子查询表示 DML 语句中使用的表
- WITH CHECK OPTION 关键字避免修改子查询范围外的数据

```
INSERT INTO (SELECT employee_id, last_name, email,
                  hire_date, job_id, salary
              FROM employees
              WHERE department_id = 50 WITH CHECK OPTION)
VALUES (99998, 'Smith', 'JSMITH',
        TO_DATE('07-JUN-99', 'DD-MON-RR'),
        'ST_CLERK', 5000);
INSERT INTO
*
```

ERROR at line 1:  
ORA-01402: view WITH CHECK OPTION where-clause violation



# 显式默认值

- 使用 **DEFAULT** 关键字表示默认值
- 可以使用显示默认值控制默认值的使用
- 显示默认值可以在 **INSERT** 和 **UPDATE** 语句中使用

# 显示使用默认值

- 在插入操作中使用默认值:

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

- 在更新操作中使用默认值:

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

# 合并语句

- 按照指定的条件执行插入或更新操作
- 如果满足条件的行存在，执行更新操作；否则执行插入操作：
  - 避免多次重复执行插入和删除操作
  - 提高效率而且使用方便
  - 在数据仓库应用中经常使用

# 合并语句的语法

可以使用**merge**语句，根据指定的条件进行插入或更新操作

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

# 合并语句举例

在对表COPY\_EMP使用merge语句，根据指定的条件从表EMPLOYEES中插入或更新数据。

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

# 合并语句举例

```
SELECT *  
FROM COPY_EMP;
```

no rows selected

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

```
SELECT *  
FROM COPY_EMP;
```

20 rows selected.

# 数据库事务

数据库事务由以下的部分组成:

- 一个或多个**DML** 语句
- 一个 **DDL** 语句
- 一个 **DCL** 语句

# 数据库事务

- 以第一个 **DML** 语句的执行作为开始
- 以下面的其中之一作为结束：
  - **COMMIT** 或 **ROLLBACK** 语句
  - **DDL** 或 **DCL** 语句（自动提交）
  - 用户会话正常结束
  - 系统异常終了

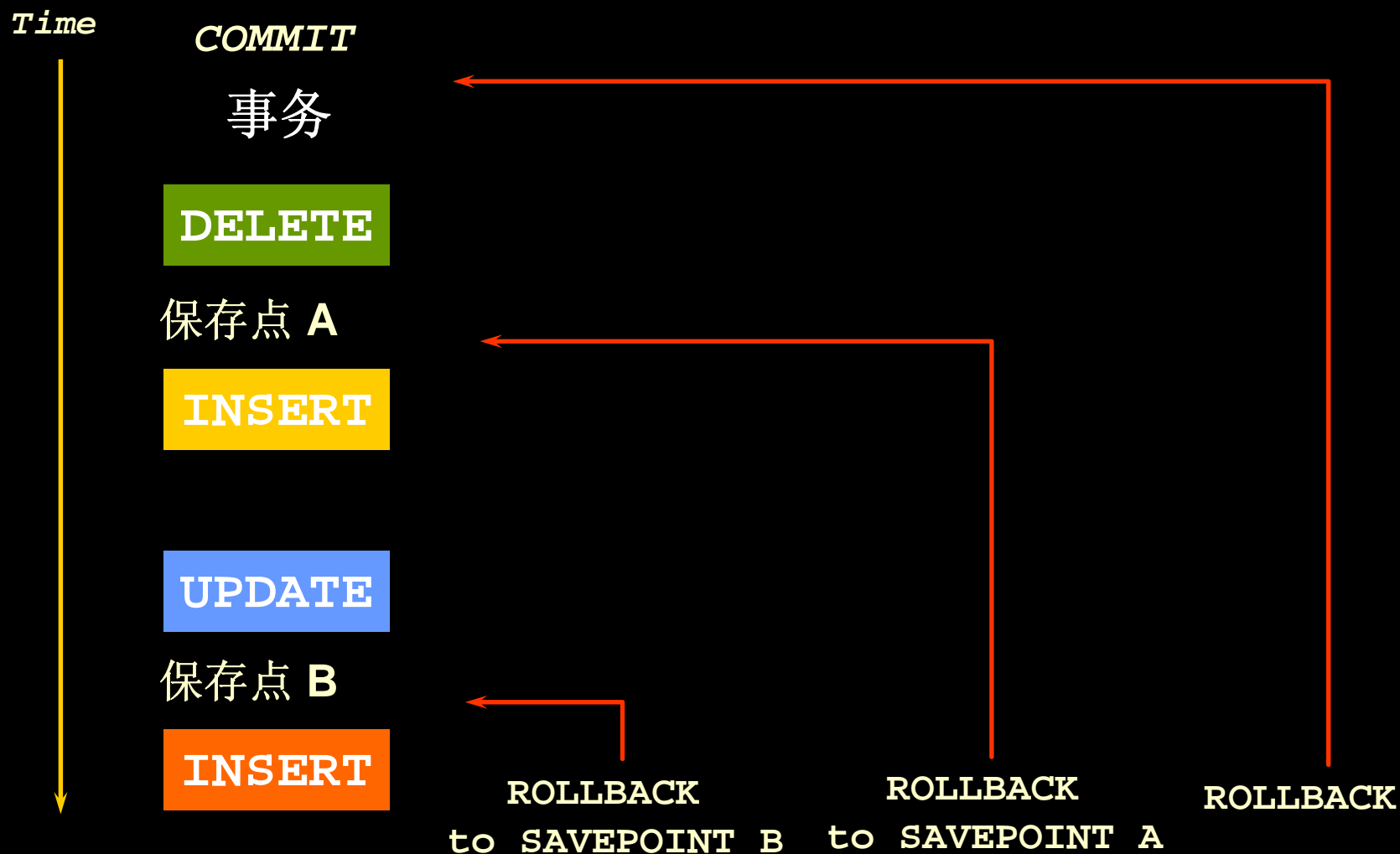


# COMMIT和ROLLBACK语句的优点

使用COMMIT 和 ROLLBACK语句,我们可以:

- 确保数据完整性。
- 数据改变被提交之前预览。
- 将逻辑上相关的操作分组。

# 控制事务



# 回滚到保留点

- 使用 **SAVEPOINT** 语句在当前事务中创建保存点。
- 使用 **ROLLBACK TO SAVEPOINT** 语句回滚到创建的保存点。

```
UPDATE...
```

```
SAVEPOINT update_done;
```

```
Savepoint created.
```

```
INSERT...
```

```
ROLLBACK TO update_done;
```

```
Rollback complete.
```

# 事务进程

- 自动提交在以下情况中执行：
  - **DDL** 语句。
  - **DCL** 语句。
  - 不使用 **COMMIT** 或 **ROLLBACK** 语句提交或回滚，正常结束会话。
- 会话异常结束或系统异常会导致自动回滚。

# 提交或回滚前的数据状态

- 改变前的数据状态是可以恢复的
- 执行 **DML** 操作的用户可以通过 **SELECT** 语句查询之前的修正
- 其他用户不能看到当前用户所做的改变，直到当前用户结束事务。
- **DML**语句所涉及到的行被锁定， 其他用户不能操作。

# 提交后的数据状态

- 数据的改变已经被保存到数据库中。
- 改变前的数据已经丢失。
- 所有用户可以看到结果。
- 锁被释放， 其他用户可以操作涉及到的数据。
- 所有保存点被释放。

# 提交数据

- 改变数据

```
DELETE FROM employees
WHERE employee_id = 99999;
1 row deleted.
```

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

- 提交改变

```
COMMIT;
Commit complete.
```

# 数据回滚后的状态

使用 **ROLLBACK** 语句可使数据变化失效:

- 数据改变被取消。
- 修改前的数据状态可以被恢复。
- 锁被释放。

```
DELETE FROM copy_emp;  
22 rows deleted.  
ROLLBACK;  
Rollback complete.
```



# 语句级回滚

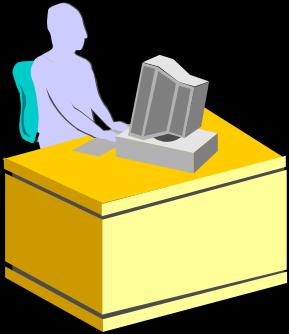
- 单独 **DML** 语句执行失败时，只有该语句被回滚。
- **Oracle** 服务器自动创建一个隐式的保留点。
- 其他数据改变仍被保留。
- 用户应执行 **COMMIT** 或 **ROLLBACK** 语句结束事务。

# 读一致性

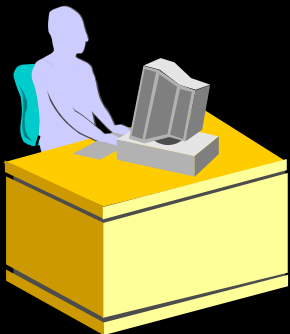
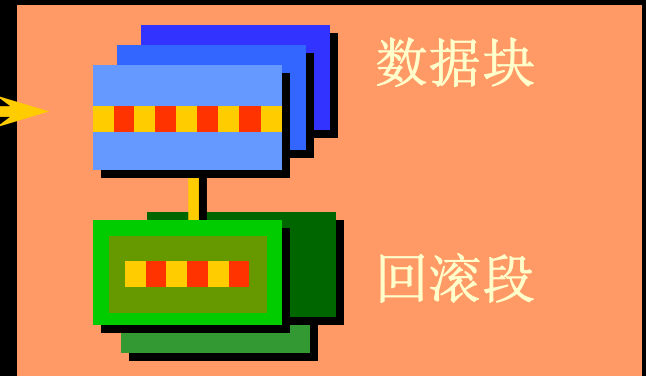
- 读一致性为数据提供一个一致的视图。
- 一个用户的对数据的改变不会影响其他用户的改变。
- 对于相同的数据读一致性保证：
  - 查询不等待修改。
  - 修改不等待查询。

# 读一致性

User A

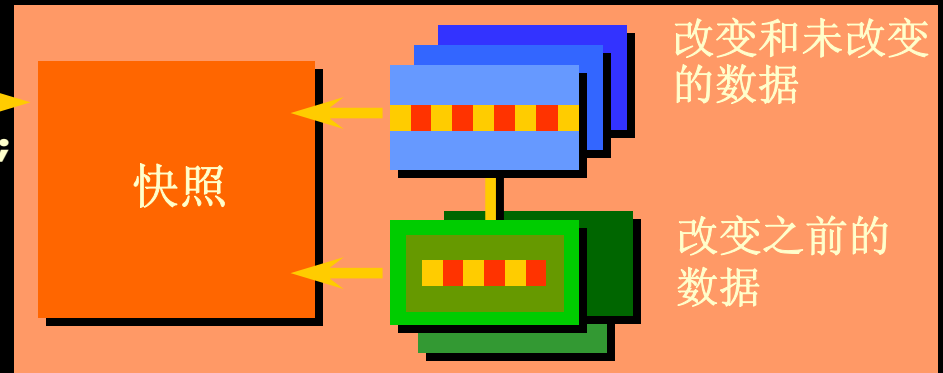


```
UPDATE employees  
SET salary = 7000  
WHERE last_name = 'Goyal';
```



User B

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



# 锁

**Oracle** 数据库中，锁是：

- 并行事务中避免资源竞争。
- 避免用户动作。
- 自动使用最低级别的限制。
- 在事务结束结束前存在。
- 两种类型：显示和隐式。

# 锁

- 两种模式：
  - 独占锁: 屏蔽其他用户。
  - 共享锁: 允许其他用户操作。
- 高级别的数据并发性：
  - **DML**: 表共享, 行独占
  - **Queries**: 不需要加锁
  - **DDL**: 保护对象定义
- 提交或回滚后锁被释放。

# 总结

通过本章学习, 您应学会如何使用**DML**语句改变数据和事务控制

语句	功能
<b>INSERT</b>	插入
<b>UPDATE</b>	修正
<b>DELETE</b>	删除
<b>MERGE</b>	合并
<b>COMMIT</b>	提交
<b>SAVEPOINT</b>	保存点
<b>ROLLBACK</b>	回滚

# 9

## 创建和管理表

# 目标

通过本章学习，您将可以：

- 描述主要的数据库对象。
- 创建表。
- 描述各种数据类型。
- 修改表的定义。
- 删除，重命名和清空表。



# 常见的数据库对象

对象	描述
表	基本的数据存储集合，由行和列组成。
视图	从表中抽出的逻辑上相关的数据集合。
序列	提供有规律的数值。
索引	提高查询的效率
同义词	给对象起别名

# 命名规则

表名和列名:

- 必须以字母开头
- 必须在 **1–30** 个字符之间
- 必须只能包含 **A–Z, a–z, 0–9, \_**, \$, 和 #
- 必须不能和用户定义的其他对象重名
- 必须 不能是**Oracle** 的保留字

# CREATE TABLE 语句

- 必须具备:
  - CREATE TABLE 权限
  - 存储空间

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

- 必须指定:
  - 表名
  - 列名, 数据类型, 尺寸

# 引用其他用户的表

- 其他用户定义的表不在当前用户的方案中
- 应该使用用户名座位前缀，引用其他用户定义的对象

# DEFAULT 选项

- 插入时为一个列指定默认值

```
... hire_date DATE DEFAULT SYSDATE, ...
```

- 字符串, 表达式, 或**SQL** 函数都是合法的
- 其它列的列名和伪列是非法的
- 默认值必须满足列的数据类型定义

# 创建表

- 语法

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

Table created.

- 确认

```
DESCRIBE dept
```

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

# Oracle 数据库中的表

- 用户定义的表:
  - 用户自己创建并维护的一组表
  - 包含了用户所需的信息
- 数据字典:
  - 由**Oracle Server**自动创建的一组表
  - 包含数据库信息

# 查询数据字典

- 查看用户定义的表.

```
SELECT table_name  
FROM user_tables ;
```

- 查看用户定义的各种数据库对象

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

- 查看用户定义的表, 视图, 同义词和序列

```
SELECT *  
FROM user_catalog ;
```



# 数据类型

数据类型	描述
<code>VARCHAR2(size)</code>	可变长字符数据
<code>CHAR(size)</code>	定长字符数据
<code>NUMBER(p,s)</code>	可变长数值数据
<code>DATE</code>	日期型数据
<code>LONG</code>	可变长字符数据，最大可达到 <b>2G</b>
<code>CLOB</code>	字符数据，最大可达到 <b>4G</b>
<code>RAW</code> and <code>LONG RAW</code>	裸二进制数据
<code>BLOB</code>	二进制数据，最大可达到 <b>4G</b>
<code>BFILE</code>	存储外部文件的二进制数据，最大可达到 <b>4G</b>
<code>ROWID</code>	行地址

# 日期数据类型

## Oracle9i对日期的改进:

- 加入了新的日期型数据类型.
- 有效的存储新数据类型.
- 提高对时区和本地时区的支持.

数据类型	描述
<b>TIMESTAMP</b>	时间撮
<b>INTERVAL YEAR TO MONTH</b>	若干年月
<b>INTERVAL DAY TO SECOND</b>	若干天到秒

# 日期数据类型

- **TIMESTAMP** 数据类型是对 **DATE** 数据类型的扩展
- 按**DATE**数据类型存放 年, 月, 日, 小时, 分钟, 秒 以及微秒甚至纳秒
- **TIMESTAMP** 数据类型的一般形式:

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

# TIMESTAMP WITH TIME ZONE

- **TIMESTAMP WITH TIME ZONE** 是一个带有时区的 **TIMESTAMP**
- 时区部分按照小时和分钟显示本地时区与**UTC**的时差

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

# TIMESTAMP WITH LOCAL TIME

- **TIMESTAMP WITH LOCAL TIME ZONE** 是一种带有本地时区的 **TIMESTAMP**
- 数据库按照数据库的本地时区存放数据
- 时区不显示在数据后面， **Oracle** 自动将数据转换为用户所在的时区
- **TIMESTAMP WITH LOCAL TIME ZONE** 的一般形式

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

# INTERVAL YEAR TO MONTH 数据

- **INTERVAL YEAR TO MONTH** 存放若干年和若干月的一个时间段。

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

```
INTERVAL '123-2' YEAR(3) TO MONTH
```

Indicates an interval of 123 years, 2 months.

```
INTERVAL '123' YEAR(3)
```

Indicates an interval of 123 years 0 months.

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

Indicates an interval of 300 months.

```
INTERVAL '123' YEAR
```

Returns an error, because the default precision is 2, and '123' has 3 digits.

# INTERVAL DAY TO SECOND 数据

- INTERVAL DAY TO SECOND 存放若干天到若干秒的一个时间段

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

```
INTERVAL '4 5:12:10.222' DAY TO SECOND(3)  
Indicates 4 days, 5 hours, 12 minutes, 10 seconds,  
and 222 thousandths of a second. INTERVAL '123' YEAR(3).
```

```
INTERVAL '7' DAY  
Indicates 7 days.
```

```
INTERVAL '180' DAY(3)  
Indicates 180 days.
```

# INTERVAL DAY TO SECOND 数据

- **INTERVAL DAY TO SECOND** 存放若干天到若干秒的一个时间段

```
INTERVAL '4 5:12:10.222' DAY TO SECOND(3)
```

Indicates 4 days, 5 hours, 12 minutes, 10 seconds, and 222 thousandths of a second.

```
INTERVAL '4 5:12' DAY TO MINUTE
```

Indicates 4 days, 5 hours and 12 minutes.

```
INTERVAL '400 5' DAY(3) TO HOUR
```

Indicates 400 days 5 hours.

```
INTERVAL '11:12:10.2222222' HOUR TO SECOND(7)
```

indicates 11 hours, 12 minutes, and 10.2222222 seconds.



# 使用子查询创建表

- 时候用 **AS subquery** 选项，将创建表和插入数据结合起来

```
CREATE TABLE table  
    [(column, column...)]  
AS subquery;
```

- 指定的列和子查询中的列要一一对应
- 通过列名和默认值定义列

# 使用子查询创建表举例

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

Table created.

```
DESCRIBE dept80
```

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

# ALTER TABLE 语句

使用 **ALTER TABLE** 语句可以:

- 追加新的列
- 修改现有的列
- 为新追加的列定义默认值
- 删除一个列

# ALTER TABLE 语句

使用 **ALTER TABLE** 语句追加, 修改, 或删除列的语法.

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

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

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

# 追加一个新列

新列

DEPT80

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

JOB_ID

追加一个新列

DEPT80

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

# 追加一个新列

- 使用 **ADD** 子句追加一个新列

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

- 新列是表中的最后一列

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

# 修改一个列

- 可以修改列的数据类型, 尺寸, 和默认值

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

- 对默认值的修改只影响今后对表的修改

# 删除一个列

使用 **DROP COLUMN** 子句删除不再需要的列.

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



# SET UNUSED 选项

- 使用 **SET UNUSED** 使一个或多个列被标记为不可用
- 使用 **DROP UNUSED COLUMNS** 选项删除不可用的列

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

OR

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

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

# 删除表

- 数据和结构都被删除
- 所有正在运行的相关事物被提交
- 所有相关索引被删除
- **DROP TABLE** 语句不能回滚

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

# 改变对象的名称

- 执行**RENAME**语句改变表, 视图, 序列, 或同义词的名称

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

- 必须是对象的拥有者

# 清空表

- **TRUNCATE TABLE** 语句:
  - 删除表中所有的数据
  - 释放表的存储空间

```
TRUNCATE TABLE detail_dept;  
Table truncated.
```

- **TRUNCATE**语句不能回滚
- 可以使用 **DELETE** 语句删除数据

# 表的注释

- 使用COMMENT 语句给表或列添加注释

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

- 可以通过下列数据字典视图查看所添加的注释:
  - ALL\_COL\_COMMENTS
  - USER\_COL\_COMMENTS
  - ALL\_TAB\_COMMENTS
  - USER\_TAB\_COMMENTS

# 总结

通过本章学习，您已经学会如何使用**DDL**语句创建, 修改, 删除, 和重命名表.

语句	描述
<b>CREATE TABLE</b>	创建表
<b>ALTER TABLE</b>	修改表结构
<b>DROP TABLE</b>	删除表
<b>RENAME</b>	重命名表
<b>TRUNCATE</b>	删除表中的所有数据，并释放存储空间
<b>COMMENT</b>	给对象加注释

# 10

约束

# 目标

通过本章学习，您将可以：

- 描述约束
- 创建和维护约束



# 什么是约束

- 约束是表级的强制规定
- 约束放置在表中删除有关联关系的数据
- 有以下五种约束：
  - NOT NULL
  - UNIQUE
  - PRIMARY KEY
  - FOREIGN KEY
  - CHECK

# 注意事项

- 如果不指定约束名 **Oracle server** 自动按照 **sys\_Cn** 的格式指定约束名
- 在什么时候创建约束:
  - 建表的同时
  - 建表之后
- 可以在表级或列级定义约束
- 可以通过数据字典视图查看约束

# 定义约束

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

```
CREATE TABLE employees(
    employee_id    NUMBER(6),
    first_name     VARCHAR2(20),
    ...
    job_id         VARCHAR2(10) NOT NULL,
    CONSTRAINT emp_emp_id_pk
        PRIMARY KEY (EMPLOYEE_ID));
```

# 定义约束

- 列级

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

- 表级

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

# NOT NULL 约束

保证列值不能为空:

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

■ ■ ■

20 rows selected.

↑  
**NOT NULL 约束**

↑  
**NOT NULL  
约束**

↑  
**无NOT NULL 约束**

# NOT NULL 约束

只能定义在列级:


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

← 系统命名

← 用户命名

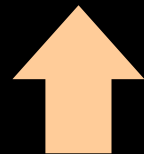
# UNIQUE 约束

EMPLOYEES



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

...



INSERT INTO

208	Smith	JSMITH
209	Smith	JSMITH



允许  
不允许: 已经存在

# UNIQUE 约束

可以定义在表级或列级:

```
CREATE TABLE employees(  
    employee_id      NUMBER(6),  
    last_name        VARCHAR2(25) NOT NULL,  
    email            VARCHAR2(25) ,  
    salary           NUMBER(8,2),  
    commission_pct   NUMBER(2,2),  
    hire_date        DATE NOT NULL,  
    ...  
    CONSTRAINT emp_email_uk UNIQUE(email));
```



# PRIMARY KEY 约束

## DEPARTMENTS

 PRIMARY KEY

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

...

不允许  
(空值)

 INSERT INTO

	Public Accounting		1400
50	Finance	124	1500

不允许  
(50 已经存在)

# PRIMARY KEY 约束

可以定义在表级或列级:

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

# FOREIGN KEY 约束

DEPARTMENTS

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

PRIMARY  
KEY



...



EMPLOYEES

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

FOREIGN  
KEY



...

INSERT INTO



200	Ford	9
201	Ford	60

不允许(9 不存在)

允许

ORACLE

# FOREIGN KEY 约束

可以定义在表级或列级:

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

# FOREIGN KEY 约束的关键字

- **FOREIGN KEY:** 在表级指定子表中的列
- **REFERENCES:** 标示在父表中的列
- **ON DELETE CASCADE:** 当父表中的列被删除是，子表中相对应的列也被删除
- **ON DELETE SET NULL:** 子表中相应的列置空

# CHECK 约束

- 定义每一行必须满足的条件
- 以下的表达式是不允许的:
  - 出现CURRVAL, NEXTVAL, LEVEL, 和ROWNUM 伪列
  - 使用 SYSDATE, UID, USER, 和 USERENV 函数
  - 在查询中涉及到其它列的值

```
..., salary    NUMBER(2)  
    CONSTRAINT emp_salary_min  
           CHECK (salary > 0),...
```

# 添加约束的语法

使用 **ALTER TABLE** 语句:

- 添加或删除约束, 但是不能修改约束
- 有效化或无效化约束
- 添加 **NOT NULL** 约束要使用 **MODIFY** 语句

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

# 添加约束

## 添加约束举例

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



# 删除约束

- 从表 **EMPLOYEES** 中删除约束

```
ALTER TABLE      employees
DROP CONSTRAINT    emp_manager_fk;
Table altered.
```

- 使用**CASCADE**选项删除约束

```
ALTER TABLE      departments
DROP PRIMARY KEY CASCADE;
Table altered.
```

# 无效化约束

- 在 **ALTER TABLE** 语句中使用 **DISABLE** 子句将约束无效化。
- 使用 **CASCADE** 选项将相关的约束也无效化

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

# 激活约束

- **ENABLE** 子句可将当前无效的约束激活

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

- 当定义或激活**UNIQUE** 或 **PRIMARY KEY** 约束时系统会自动创建**UNIQUE** 或 **PRIMARY KEY**索引

## 及连约束

- **CASCADE CONSTRAINTS** 子句在 **DROP COLUMN** 子句中使用
- 在删除表的列时 **CASCADE CONSTRAINTS** 子句指定将相关的约束一起删除
- 在删除表的列时 **CASCADE CONSTRAINTS** 子句同时也删除多列约束

# 及连约束

及连约束举例:

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

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

# 查询约束

## 查询数据字典视图 USER\_CONSTRAINTS

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

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

...

# 查询定义约束的列

## 查询数据字典视图 USER\_CONS\_COLUMNS

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

CONSTRAINT_NAME	COLUMN_NAME
EMP_DEPT_FK	DEPARTMENT_ID
EMP_EMAIL_NN	EMAIL
EMP_EMAIL_UK	EMAIL
EMP_EMP_ID_PK	EMPLOYEE_ID
EMP_HIRE_DATE_NN	HIRE_DATE
EMP_JOB_FK	JOB_ID
EMP_JOB_NN	JOB_ID

...

# 总结

通过本章学习，您已经学会如何创建约束  
描述约束的类型：

- NOT NULL
  - UNIQUE
  - PRIMARY KEY
  - FOREIGN KEY
  - CHECK
- 查询数据字典视图 以获得约束的信息





# 11

视图

# 目标

通过本章学习，您将可以：

- 描述视图
- 创建和修改视图的定义，删除视图
- 从视图中查询数据
- 通过视图插入, 修改和删除数据
- 创建和使用临时视图
- 使用“**Top-N**” 分析

# 常见的数据库对象

对象	描述
表	基本的数据存储集合，由行和列组成。
视图	从表中抽出的逻辑上相关的数据集合。
序列	提供有规律的数值。
索引	提高查询的效率
同义词	给对象起别名

# 视图

表EMPLOYEES :

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	4200
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	5800
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	3500
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	3100
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	2600
149	Zlotkey				JUL-98	ST_CLERK	2500
174	Abel				JAN-00	SA_MAN	10500
176	Taylor				MAY-96	SA_REP	11000
176	Kimberely	Grant	KGRANT	515.144.1644, 425263	MAR-98	SA_REP	8600
176	Kimberely	Grant	KGRANT	515.144.1644, 425263	24-MAY-99	SA_REP	7000
200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	4400
201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-96	MK_MAN	13000
202	Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK_REP	6000
205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	12000
206	William	Gietz	WGIEZT	515.123.8181	07-JUN-94	AC_ACCOUNT	8300

20 rows selected.

# 为什么使用视图

- 控制数据访问
- 简化查询
- 数据独立性
- 避免重复访问相同的数据

# 简单视图和复杂视图

特性	简单视图	复杂视图
表的数量	一个	一个或多个
函数	没有	有
分组	没有	有
<b>DML</b> 操作	可以	有时可以

# 创建视图

- 在CREATE VIEW语句中嵌入子查询

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view  
  [(alias[, alias]...)]  
  AS subquery  
[WITH CHECK OPTION [CONSTRAINT constraint]]  
[WITH READ ONLY [CONSTRAINT constraint]];
```

- 子查询可以是复杂的 SELECT 语句



# 创建视图

- 创建视图举例

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

View created.

- 描述视图结构

```
DESCRIBE empvu80
```

# 创建视图

- 创建视图时在子查询中给列定义别名

```
CREATE VIEW  salvu50
  AS SELECT  employee_id ID_NUMBER, last_name NAME,
            salary*12 ANN_SALARY
    FROM      employees
   WHERE      department_id = 50;
```

View created.

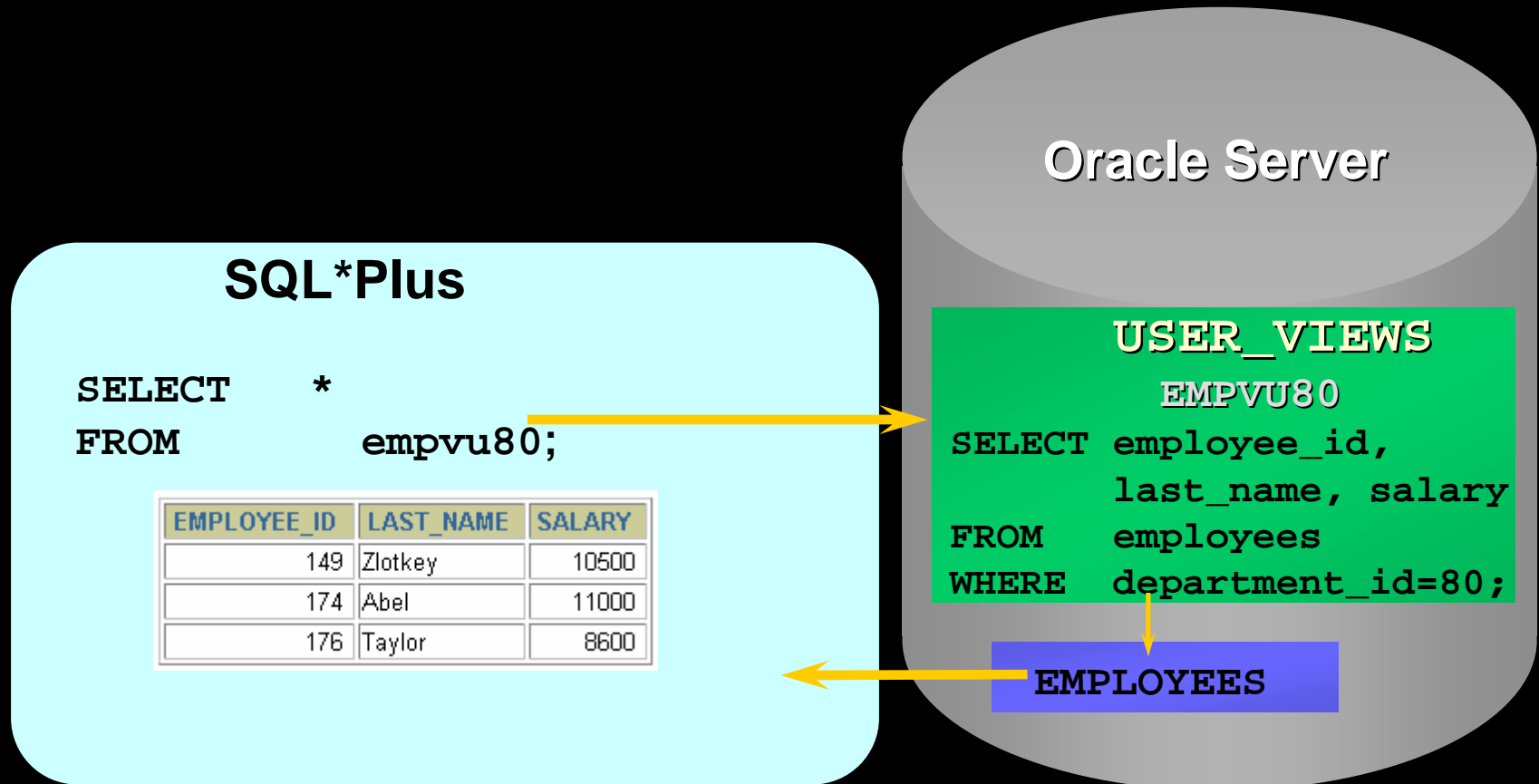
- 在选择视图中的列时应使用别名

# 查询视图

```
SELECT *  
FROM salvu50;
```

ID_NUMBER	NAME	ANN_SALARY
124	Mourgos	69600
141	Rajs	42000
142	Davies	37200
143	Matos	31200
144	Vargas	30000

# 查询视图



# 修改视图

- 使用CREATE OR REPLACE VIEW 子句修改视图

```
CREATE OR REPLACE VIEW empvu80
  (id_number, name, sal, department_id)
AS SELECT  employee_id, first_name || ' ' || last_name,
           salary, department_id
  FROM      employees
 WHERE     department_id = 80;
```

View created.

- CREATE VIEW 子句中各列的别名应和子查询中各列相对应

# 创建复杂视图

## 复杂视图举例

```
CREATE VIEW dept_sum_vu
  (name, minsal, maxsal, avgsal)
AS SELECT      d.department_name, MIN(e.salary),
               MAX(e.salary),AVG(e.salary)
  FROM          employees e, departments d
  WHERE         e.department_id = d.department_id
  GROUP BY     d.department_name;
```

View created.

# 视图中使用DML的规定

- 可以在简单视图中执行 **DML** 操作
- 当视图定义中包含以下元素之一时不能使用**delete**:
  - 组函数
  - **GROUP BY** 子句
  - **DISTINCT** 关键字
  - **ROWNUM** 伪列

# 视图中使用**DML**的规定

当视图定义中包含以下元素之一时不能使用**update** :

- 组函数
- **GROUP BY**子句
- **DISTINCT** 关键字
- **ROWNUM** 伪列
- 列的定义为表达式



# 视图中使用**DML**的规定

当视图定义中包含以下元素之一时不能使用**insert** :

- 组函数
- **GROUP BY** 子句
- **DISTINCT** 关键字
- **ROWNUM** 伪列
- 列的定义为表达式
- 表中非空的列在视图定义中未包括

## WITH CHECK OPTION 子句

- 使用 **WITH CHECK OPTION** 子句确保**DML**只能在特定的范围内执行

```
CREATE OR REPLACE VIEW empvu20
AS SELECT      *
   FROM        employees
   WHERE       department_id = 20
   WITH CHECK OPTION CONSTRAINT empvu20_ck ;
```

View created.

- 任何违反**WITH CHECK OPTION** 约束的请求都会失败

# 屏蔽 DML 操作

- 可以使用 **WITH READ ONLY** 选项屏蔽对视图的**DML** 操作
- 任何 **DML** 操作都会返回一个**Oracle server** 错误

# 屏蔽 DML 操作

```
CREATE OR REPLACE VIEW empvu10  
  (employee_number, employee_name, job_title)  
AS SELECT  employee_id, last_name, job_id  
  FROM      employees  
  WHERE     department_id = 10  
  WITH READ ONLY;
```

View created.

# 删除视图

删除视图只是删除视图的定义，并不会删除基表的数据

```
DROP VIEW view;
```

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

# 临时视图

- 临时视图可以是嵌套在 **SQL**语句中的子查询
- 在**FROM** 子句中的子查询是临时视图
- 临时视图不是数据库对象

# Top-N 分析

- **Top-N** 分析查询一个列中最大或最小的  $n$  个值:
  - 销售量最高的十种产品是什么?
  - 销售量最差的十种产品是什么?
- 最大和最小的值的集合是 **Top-N** 分析所关心的

# Top-N 分析

查询最大的几个值的 **Top-N** 分析:

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



# Top-N 分析

查询工资最高的三名员工:

1 2 3

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

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

1 2 3

# 总结

通过本章学习，您已经了解视图的优点和基本应用：

- 控制数据访问
- 简化查询
- 数据独立性
- 删除时不删除数据
- 子查询是临时视图的一种
- **Top-N** 分析

# 12

其它数据库对象

# 目标

通过本章学习，您将可以：

- 创建, 维护, 和使用序列
- 创建和维护索引
- 创建私有和公有同义词

# 常见的数据库对象

对象	描述
表	基本的数据存储集合，由行和列组成。
视图	从表中抽出的逻辑上相关的数据集合。
序列	提供有规律的数值。
索引	提高查询的效率
同义词	给对象起别名

# 什么是序列？

序列：

- 自动提供唯一的数值
- 共享对象
- 主要用于提供主键值
- 代替应用代码
- 将序列值装入内存可以提高访问效率

# CREATE SEQUENCE 语句

定义序列:

```
CREATE SEQUENCE sequence
    [INCREMENT BY n]
    [START WITH n]
    [{MAXVALUE n | NOMAXVALUE}]
    [{MINVALUE n | NOMINVALUE}]
    [{CYCLE | NOCYCLE}]
    [{CACHE n | NOCACHE}] ;
```

# 创建序列

- 创建序列 DEPT\_DEPTID\_SEQ 为表 DEPARTMENTS 提供主键
- 不使用 CYCLE 选项

```
CREATE SEQUENCE dept_deptid_seq  
        INCREMENT BY 10  
        START WITH 120  
        MAXVALUE 9999  
        NOCACHE  
        NOCYCLE;
```

**Sequence created.**



# 查询序列

- 查询数据字典视图 **USER\_SEQUENCES** 获取序列定义信息

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

- 如果指定 **NOCACHE** 选项，则列 **LAST\_NUMBER** 显示序列中下一个有效的值

## NEXTVAL 和 CURRVAL 伪列

- **NEXTVAL** 返回序列中下一个有效的值，任何用户都可以引用
- **CURRVAL** 中存放序列的当前值
- **NEXTVAL** 应在 **CURRVAL** 之前指定，二者应同时有效

# 序列应用举例

```
INSERT INTO departments(department_id,  
                        department_name, location_id)  
VALUES      (dept_deptid_seq.NEXTVAL,  
            'Support', 2500);  
  
1 row created.
```

- 序列 DEPT\_DEPTID\_SEQ 的当前值

```
SELECT      dept_deptid_seq.CURRVAL  
FROM        dual;
```

# 使用序列

- 将序列值装入内存可提高访问效率
- 序列在下列情况下出现裂缝:
  - 回滚
  - 系统异常
  - 多个表同时使用同一序列
- 如果不讲序列的值装入内存(**NOCACHE**), 可使用表 **USER\_SEQUENCES** 查看序列当前的有效值

# 修改序列

修改序列的增量, 最大值, 最小值, 循环选项, 或是否装入内存

```
ALTER SEQUENCE dept_deptid_seq  
        INCREMENT BY 20  
        MAXVALUE 999999  
        NOCACHE  
        NOCYCLE;
```

**Sequence altered.**

# 修改序列的注意事项

- 必须是序列的拥有者或对序列有 **ALTER** 权限
- 只有将来的序列值会被改变
- 改变序列的初始值只能通过删除序列之后重建序列的方法实现
- 其它的一些限制

# 删除序列

- 使用 **DROP SEQUENCE** 语句删除序列
- 删除之后，序列不能再次被引用

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

# 索引

## 索引:

- 一种数据库对象
- 通过指针加速 **Oracle** 服务器的查询速度
- 通过快速定位数据的方法，减少磁盘 I/O
- 索引与表相互独立
- **Oracle** 服务器自动使用和维护索引



# 创建索引

- 自动创建: 在定义 **PRIMARY KEY** 或 **UNIQUE** 约束后系统自动在相应的列上创建唯一性索引
- 手动创建: 用户可以在其它列上创建非唯一的索引, 以加速查询

# 创建索引

- 在一个或多个列上创建索引

```
CREATE INDEX index  
ON table (column[, column]...);
```

- 在表 EMPLOYEES 的列 LAST\_NAME 上创建索引

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

# 什么时候创建索引

以下情况可以创建索引:

- 列中数据值分布范围很广
- 列中包含大量空值
- 列经常在 **WHERE** 子句或连接条件中出现
- 表经常被访问而且数据量很大，访问的数据大概占数据总量的**2%到4%**

# 什么时候不要创建索引

下列情况不要创建索引:

- 表很小
- 列不经常作为连接条件或出现在**WHERE**子句中
- 查询的数据大于**2%到4%**
- 表经常更新
- 加索引的列包含在表达式中

# 查询索引

- 可以使用数据字典视图USER\_INDEXES 和 USER\_IND\_COLUMNS 查看索引的信息

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

# 基于函数的索引

- 基于函数的索引是一个基于表达式的索引
- 索引表达式由列, 常量, **SQL** 函数和用户自定义的函数

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

Index created.

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

# 删除索引

- 使用DROP INDEX 命令删除索引

```
DROP INDEX index;
```

- 删除索引|UPPER\_LAST\_NAME\_IDX

```
DROP INDEX upper_last_name_idx;  
Index dropped.
```

- 只有索引的拥有者或拥有DROP ANY INDEX权限的用户才可以删除索引

# 同义词

使用同义词访问相同的对象:

- 方便访问其它用户的对象
- 缩短对象名字的长度

```
CREATE [PUBLIC] SYNONYM synonym  
FOR    object;
```



# 创建和删除同义词

- 为视图DEPT\_SUM\_VU 创建同义词

```
CREATE SYNONYM d_sum  
FOR dept_sum_vu;  
Synonym Created.
```

- 删除同义词

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

# 总结

通过本章学习,您已经可以:

- 使用序列
- 通过数据字典视图 **USER\_SEQUENCES** 查看序列信息
- 使用索引提高查询效率
- 通过数据字典视图 **USER\_INDEXES** 查看索引信息
- 为数据对象定义同义词

# 13

## 控制用户权限

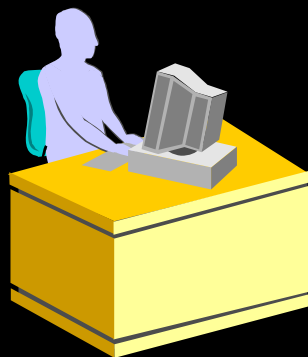
# 目标

通过本章学习，您将可以：

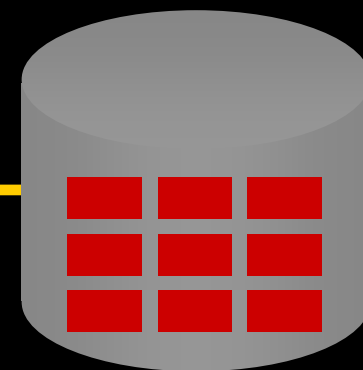
- 创建用户
- 创建角色
- 使用 **GRANT** 和 **REVOKE** 语句赋予和回收权限
- 创建数据库联接

# 控制用户权限

数据库管理员



用户名和密码  
权限



用户



# 权限

- 数据库安全性:
  - 系统安全性
  - 数据安全性
- 系统权限: 对于数据库的权限
- 对象权限: 操作数据库对象的权限
- 方案: 一组数据库对象集合, 例如表, 视图, 和序列

# 系统权限

- 超过一百多种 **100** 有效的权限
- 数据库管理员具有高级权限以完成管理任务，例如：
  - 创建新用户
  - 删除用户
  - 删除表
  - 备份表

# 创建用户

**DBA 使用 CREATE USER 语句创建用户**

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

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



# 用户的系统权限

- 用户创建之后, **DBA** 会赋予用户一些系统权限

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

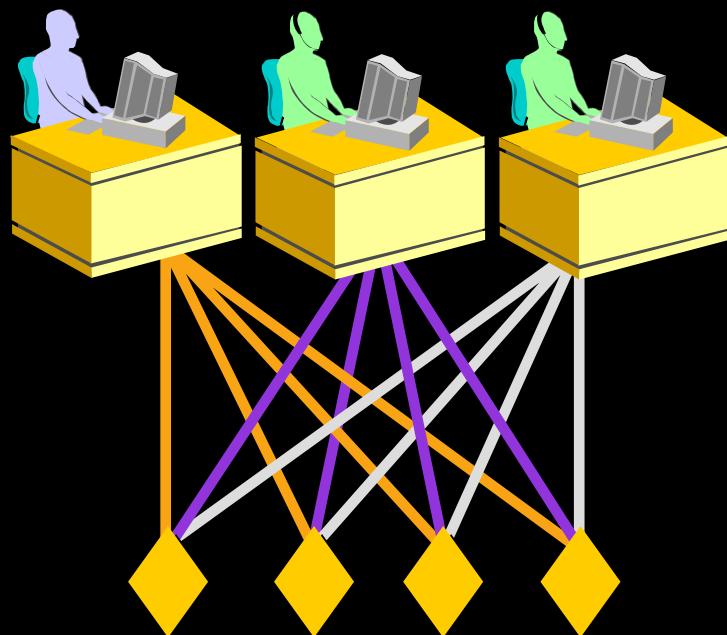
- 以应用程序开发者为例, 一般具有下列系统权限:
  - **CREATE SESSION** (创建会话)
  - **CREATE TABLE** (创建表)
  - **CREATE SEQUENCE** (创建序列)
  - **CREATE VIEW** (创建视图)
  - **CREATE PROCEDURE** (创建过程)

# 赋予系统权限

**DBA** 可以赋予用户特定的权限

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

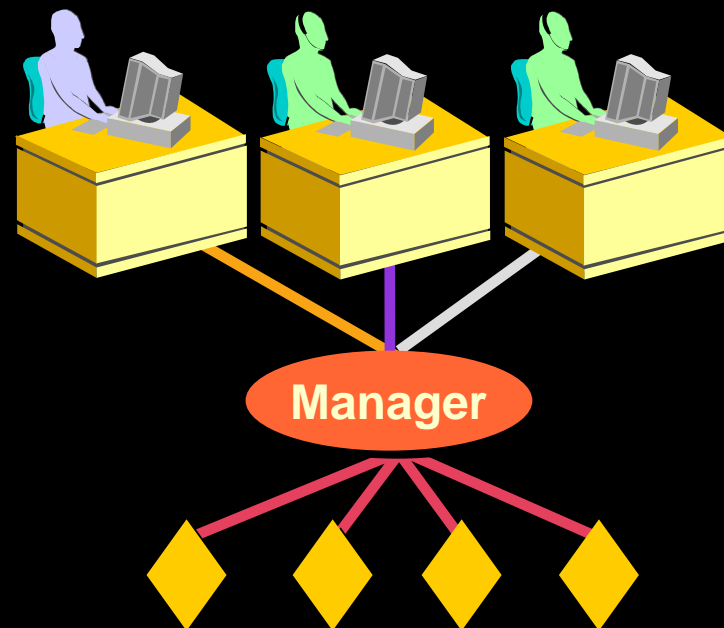
# 角色



不使用角色分配权限

用户

权限



使用角色分配权限

# 创建角色并赋予权限

- 创建角色

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

- 为角色赋予权限

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

- 将角色赋予用户

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

# 修改密码

- **DBA** 可以创建用户和修改密码
- 用户本人可以使用 **ALTER USER** 语句修改密码

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

# 对象权限

对象权限	表	视图	序列	过程
修改	√		√	
删除	√	√		
执行				√
索引	√			
插入	√	√		
关联	√	√		
选择	√	√	√	
更新	√	√		

# 对象权限

- 不同的对象具有不同的对象权限
- 对象的拥有者拥有所有权限
- 对象的拥有者可以向外分配权限

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

# 分配对象权限

- 分配表 **EMPLOYEES** 的查询权限

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

- 分配表中各个列的更新权限

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



# WITH GRANT OPTION 和 PUBLIC 关键字

- WITH GRANT OPTION 使用户同样具有分配权限的权利

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

- 向数据库中所有用户分配权限

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

# 查询权限分配情况

数据字典视图	描述
<b>ROLE_SYS_PRIVS</b>	角色拥有的系统权限
<b>ROLE_TAB_PRIVS</b>	角色拥有的对象权限
<b>USER_ROLE_PRIVS</b>	用户拥有的角色
<b>USER_TAB_PRIVS_MADE</b>	用户分配的关于表对象权限
<b>USER_TAB_PRIVS_RECD</b>	用户拥有的关于表对象权限
<b>USER_COL_PRIVS_MADE</b>	用户分配的关于列的对象权限
<b>USER_COL_PRIVS_RECD</b>	用户拥有的关于列的对象权限
<b>USER_SYS_PRIVS</b>	用户拥有的系统权限

# 收回对象权限

- 使用 **REVOKE** 语句收回权限
- 使用 **WITH GRANT OPTION** 子句所分配的权限同样被收回

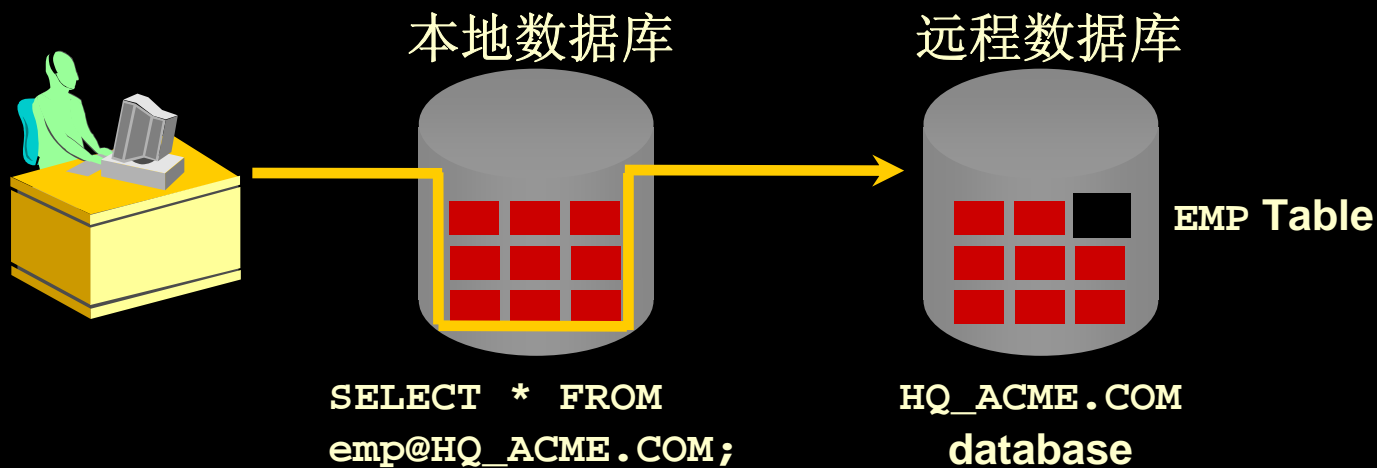
```
REVOKE {privilege [, privilege...]|ALL}  
ON      object  
FROM    {user[, user...]|role|PUBLIC}  
[CASCADE CONSTRAINTS];
```

# 收回对象权限举例

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

# 数据库联接

数据库联接使用户可以在本地访问远程数据库



# 数据库联接

- 创建数据库联接

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

- 使用SQL 语句访问远程数据库

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

# 总结

通过本章学习,您已经可以使用 **DCL** 控制数据库权限,创建数据库联接:

语句	功能
<b>CREATE USER</b>	创建用户 (通常由 <b>DBA</b> 完成)
<b>GRANT</b>	分配权限
<b>CREATE ROLE</b>	创建角色 (通常由 <b>DBA</b> 完成)
<b>ALTER USER</b>	修改用户密码
<b>REVOKE</b>	收回权限

# 15

## SET 运算符

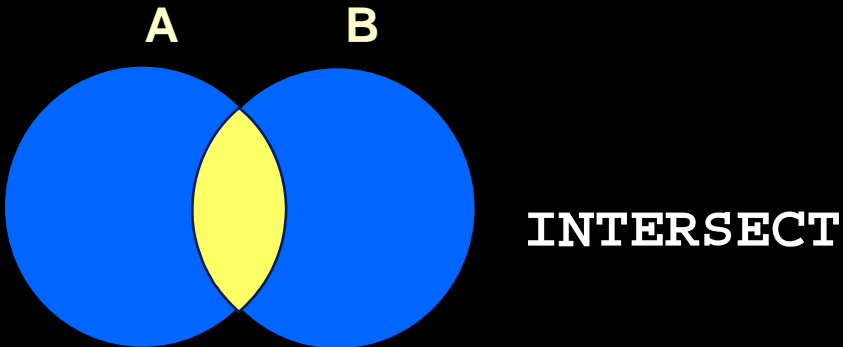
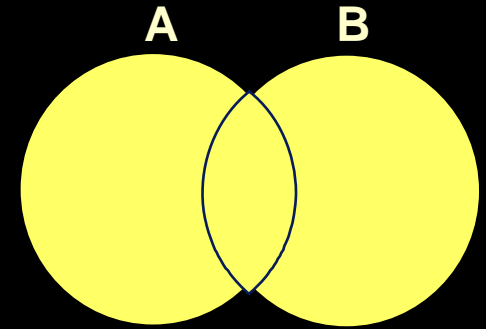
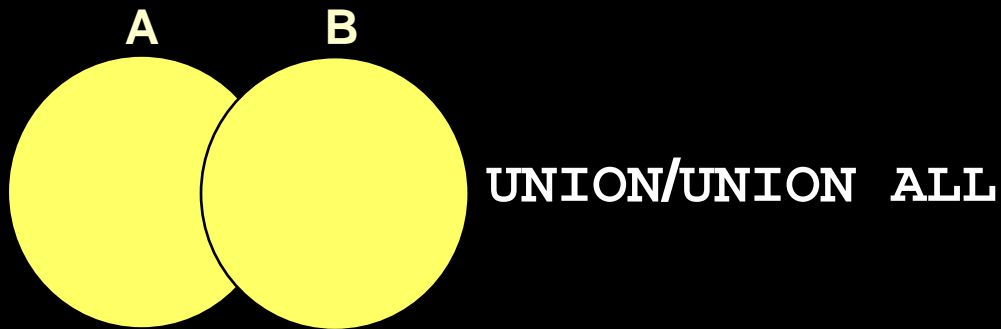


# 目标

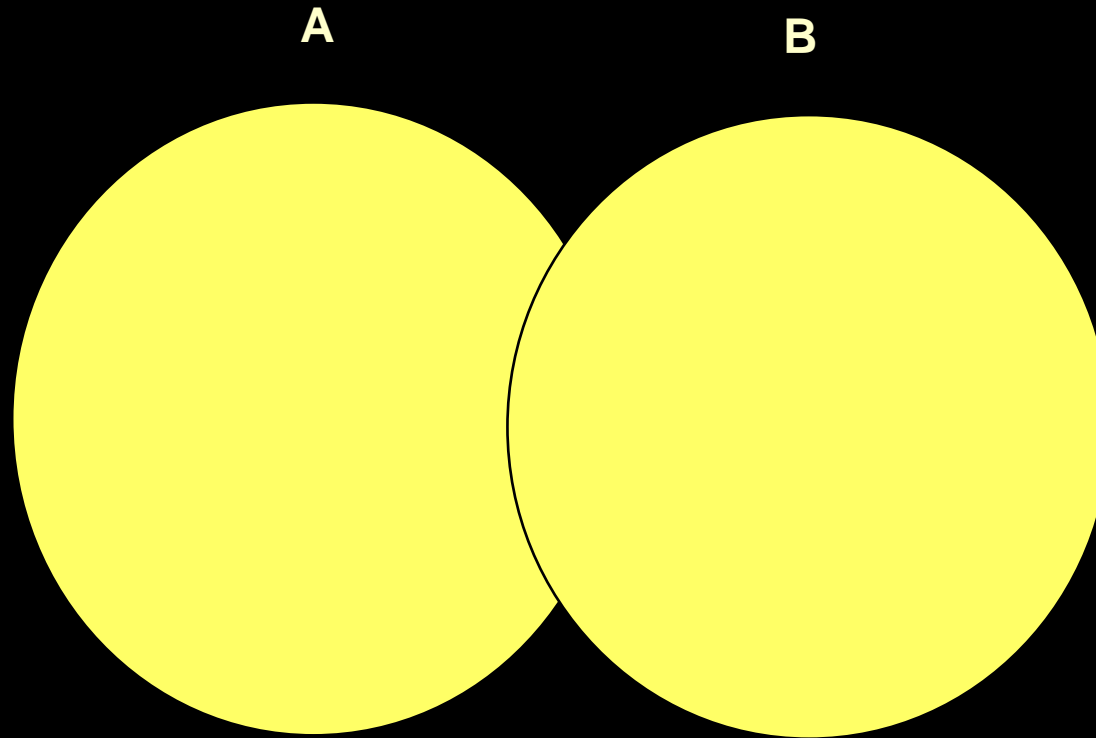
通过本章学习，您将可以：

- 描述 **SET** 操作符
- 将多个查询用**SET** 操作符联接组成一个新的查询
- 排序

# SET 操作符



# UNION 操作符



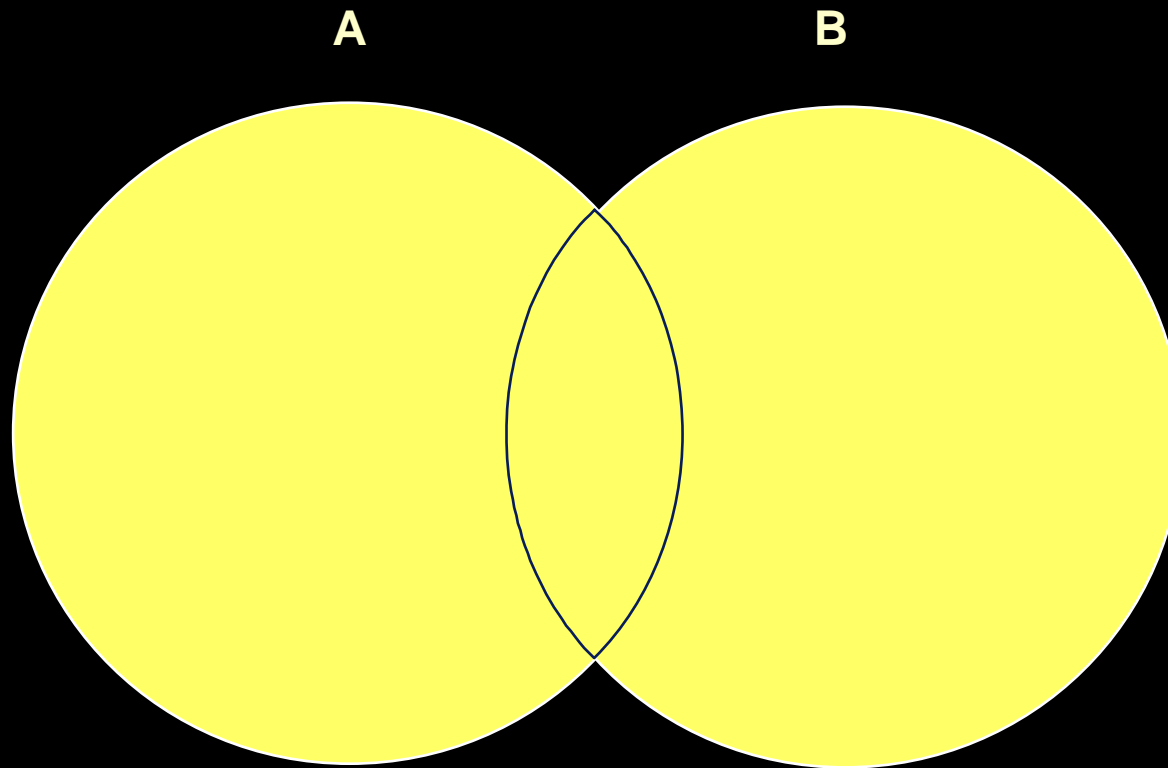
UNION 操作符返回两个查询的结果集的并集

# UNION 操作符举例

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

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

# UNION ALL 操作符



**UNION ALL** 操作符返回两个查询的结果集的并集以及两个结果集的重复部分（不去重）

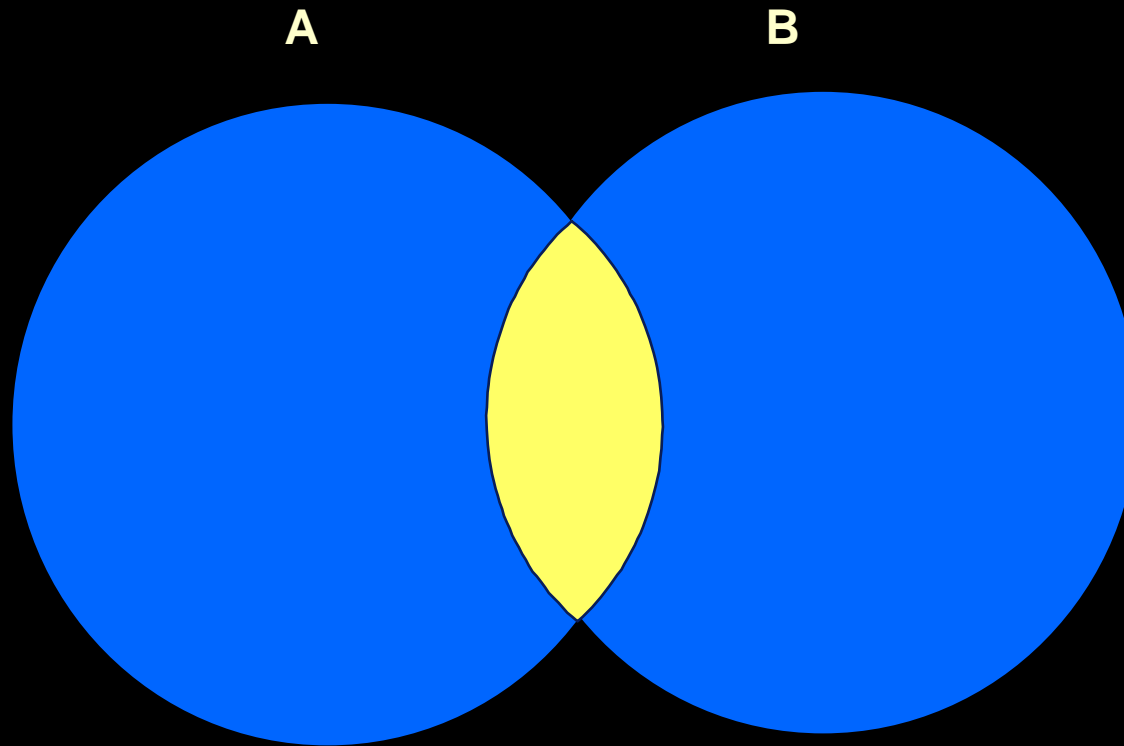
# UNION ALL 操作符举例

```
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.

# INTERSECT 操作符



**INTERSECT** 操作符返回两个结果集的交集

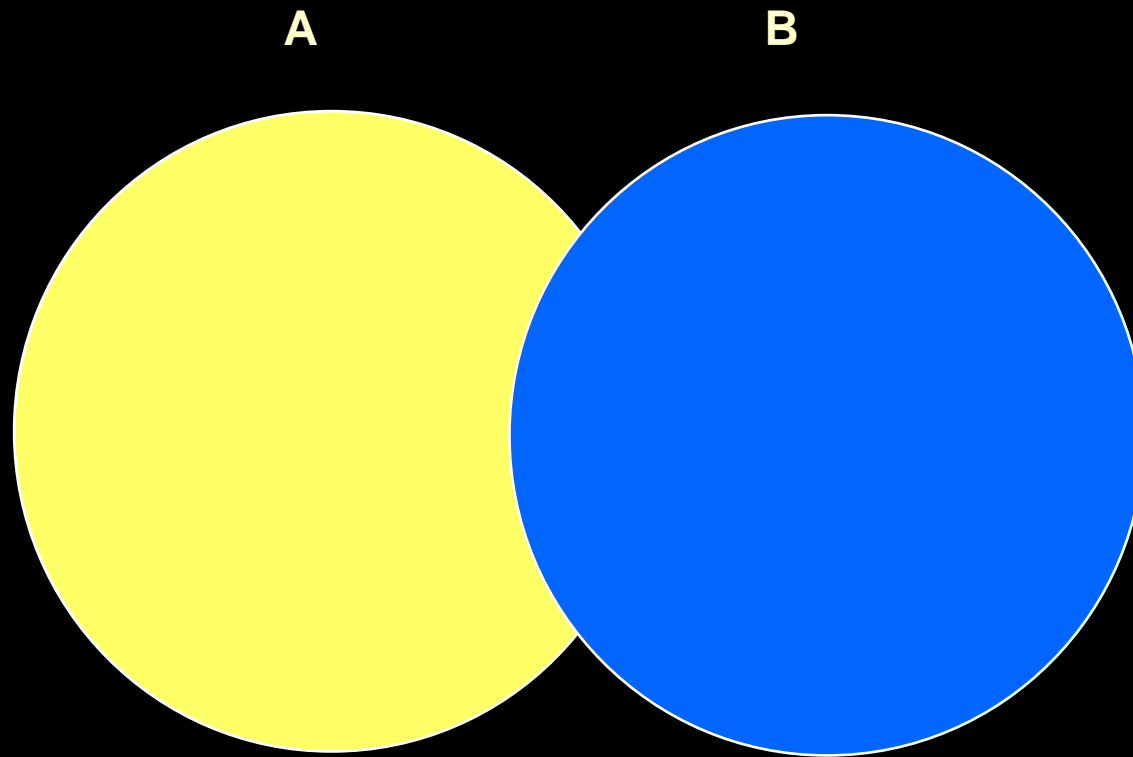
# INTERSECT 操作符举例

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

EMPLOYEE_ID	JOB_ID
176	SA_REP
200	AD_ASST



# MINUS 操作符



**MINUS** 操作符返回两个结果集的补集

# MINUS 操作符举例

```
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.

# 使用 **SET** 操作符注意事项

- 在**SELECT** 列表中的列名和表达式在数量和数据类型上要相对应
- 括号可以改变执行的顺序
- **ORDER BY** 子句:
  - 只能在语句的最后出现
  - 可以使用第一个查询中的列名, 别名或相对位置

# SET 操作符

- 除 **UNION ALL** 之外，系统会自动将重复的记录删除
- 系统将第一个查询的列名显示在输出中
- 除 **UNION ALL** 之外，系统自动按照第一个查询中的第一个列的升序排列

# 匹配各SELECT 语句举例

```
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.

## 匹配各SELECT 语句举例

```
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.

# 使用相对位置排序举例

```
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

# 总结

通过本章学习,您已经可以:

- 使用 **UNION** 操作符
- 使用 **UNION ALL** 操作符
- 使用 **INTERSECT** 操作符
- 使用 **MINUS**操作符
- 使用 **ORDER BY** 对结果集排序



# Practice 15 Overview

**This practice covers using the Oracle9i datetime functions.**

# 16

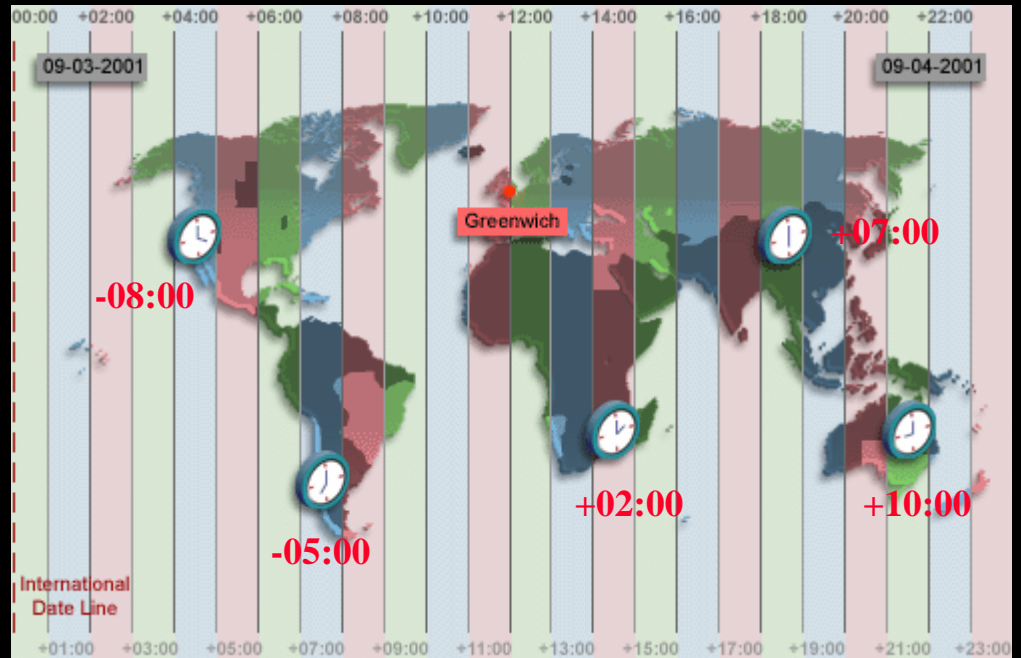
## Oracle9i 日期函数

# 目标

通过本章学习，您将可以使用下列日期函数：

- **TZ\_OFFSET**
- **CURRENT\_DATE**
- **CURRENT\_TIMESTAMP**
- **LOCALTIMESTAMP**
- **DBTIMEZONE**
- **SESSIONTIMEZONE**
- **EXTRACT**
- **FROM\_TZ**
- **TO\_TIMESTAMP**
- **TO\_TIMESTAMP\_TZ**
- **TO\_YMINTERVAL**

# 时区



上图显示了全球**24**个时区以及当格林威治时间是 **12:00**时各时区的时差

# Oracle9i 日期支持

- **Oracle9i**中, 可以将时区加入到日期和时间中而且可以将秒进行进一步的精确
- 日期中加入了三种新的数据类型:
  - **TIMESTAMP** (时间撮)
  - **TIMESTAMP WITH TIME ZONE (TSTZ)** (带时区的时间撮)
  - **TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ)** (带有本地时区的时间撮)
- **Oracle9i** 支持夏令时

## TZ\_OFFSET

- 显示时区 'US/Eastern' 的时差

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

TZ_OFFSET
-04:00

- 显示时区 'Canada/Yukon' 的时差

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

TZ_OFFSET
-07:00

- 显示时区 'Europe/London' 的时差

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

TZ_OFFSET
+01:00

# CURRENT\_DATE

- 按照当前会话的时区显示当前会话的时间

```
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 对会话所在的时区是敏感的

# CURRENT\_TIMESTAMP

- 按照当前会话的时区显示当前会话的时间

```
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** 对会话所在的时区是敏感的
- 返回值是 **TIMESTAMP WITH TIME ZONE** 数据类型



# LOCALTIMESTAMP

- 按照当前会话的时区显示当前会话的时间

```
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对会话所在的时区是敏感的
- 返回值是 **TIMESTAMP** 数据类型

# DBTIMEZONE 和 SESSIONTIMEZONE

- 显示数据库所在的时区

```
SELECT DBTIMEZONE FROM DUAL;
```

DBTIME
-05:00

- 显示会话所在的时区

```
SELECT SESSIONTIMEZONE FROM DUAL;
```

SESSIONTIMEZONE
-08:00

# EXTRACT

- 从 **SYSDATE** 中抽出年

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

EXTRACT(YEARFROMSYSDATE)
2001

- 从 **HIRE\_DATE** 中抽出月

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

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

# FROM\_TZ 应用举例

```
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
```

```
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
```

# TO\_TIMESTAMP 和 TO\_TIMESTAMP\_TZ 应用 举例

```
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

```
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

## TO\_YMINTERVAL 应用举例

```
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

# 总结

通过本章学习,您已经可以使用:

- **TZ\_OFFSET**
- **FROM\_TZ**
- **TO\_TIMESTAMP**
- **TO\_TIMESTAMP\_TZ**
- **TO\_YMINTERVAL**
- **CURRENT\_DATE**
- **CURRENT\_TIMESTAMP**
- **LOCALTIMESTAMP**
- **DBTIMEZONE**
- **SESSIONTIMEZONE**
- **EXTRACT**

# 17

## 对 GROUP BY 子句的扩展



# 目标

通过本章学习，您将可以：

- 使用 **ROLLUP** 操作分组
- 使用 **CUBE** 操作分组
- 使用 **GROUPING** 函数处理 **ROLLUP** 或 **CUBE**操作所产生的空值
- 使用 **GROUPING SETS** 操作进行单独分组

# 组函数

组函数处理多行返回一个行

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

例子:

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

# GROUP BY 子句

语法:

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

例子:

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

# HAVING 子句

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

- 使用 **HAVING** 对组函数进行限制
- 对查询进行第二次限制

## 帶有ROLLUP 和 CUBE 操作的GROUP BY 子句

- 使用帶有ROLLUP 和 CUBE 操作的GROUP BY 子句产生多种分组结果
- ROLLUP 产生 $n + 1$ 种分组结果
- CUBE 产生 $2^n$ 种分组结果

# ROLLUP 操作符

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

- ROLLUP 是对 GROUP BY 子句的扩展
- ROLLUP 产生  $n + 1$  种分组结果，顺序是从右向左

# ROLLUP 应用举例

```
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.

# CUBE 操作符

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

- CUBE是对 GROUP BY 子句的扩展
- CUBE 会产生类似于笛卡尔集的分组结果



# CUBE 应用举例

```
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.

1

2

3

4

# GROUPING 函数

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

- GROUPING 函数可以和 CUBE 或 ROLLUP 结合使用
- 使用 GROUPING 函数, 可以找到哪些列在该行中参加了分组
- 使用 GROUPING 函数, 可以区分空值产生的原因
- GROUPING 函数返回 0 或 1

# GROUPING 函数举例

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

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

6 rows selected.

# GROUPING SETS

- **GROUPING SETS** 是对 **GROUP BY** 子句的进一步扩充
- 使用 **GROUPING SETS** 在同一个查询中定义多个分组集
- **Oracle** 对 **GROUPING SETS** 子句指定的分组集进行分组后用 **UNION ALL** 操作将各分组结果结合起来
- **Grouping set** 的优点:
  - 只进行一次分组即可
  - 不必书写复杂的 **UNION** 语句
  - **GROUPING SETS** 中包含的分组项越多,性能越好。

# GROUPING SETS应用举例

```
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

1

2

26 rows selected.

# 复合列

- 复合列是被作为整体处理的一组列的集合  
`ROLLUP (a, (b, c), d)`
- 使用括号将若干列组成复合列在 `ROLLUP` 或 `CUBE` 中作为整体进行操作
- 在 `ROLLUP` 或 `CUBE` 中, 复合列可以避免产生不必要的分组结果

# 复合列应用举例

```
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.

# 连接分组集

- 连接分组集可以产生有用的对分组项的结合
- 将各分组集, **ROLLUP** 和 **CUBE** 用逗号连接 **Oracle** 自动在 **GROUP BY** 子句中将各分组集进行连接
- 连接的结果是对各分组生成笛卡尔集

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



# 连接分组集应用举例

```
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.

# 总结

通过本章学习,您已经可以:

- 使用 **ROLLUP** 操作符
- 使用 **CUBE** 操作符
- 使用 **GROUPING** 函数处理在 **ROLLUP** 或 **CUBE**中产生的空值
- 使用 **GROUPING SETS** 创建分组集
- 在 **GROUP BY** 子句中组合分组:
  - 复合列
  - 连接分组集

# 18

## 高级子查询

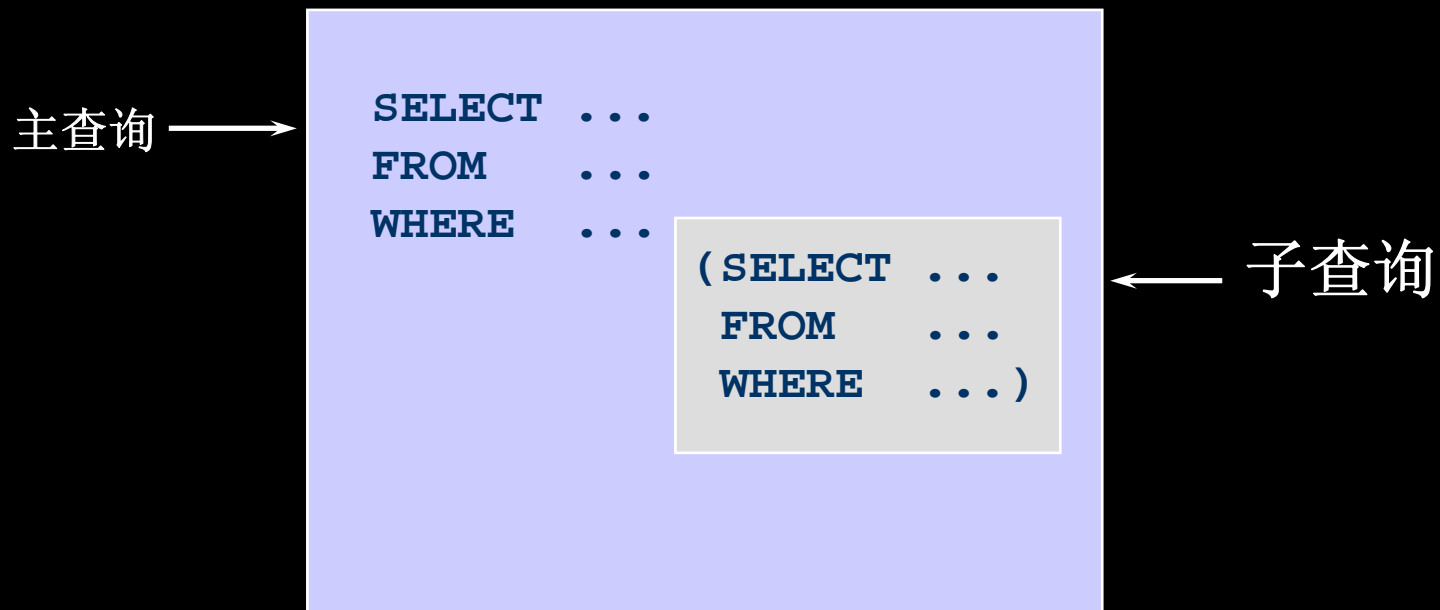
# 目标

通过本章学习，您将可以：

- 书写多列子查询
- 子查询对空值的处理
- 在 **FROM** 子句中使用子查询
- 在**SQL**中使用单列子查询
- 相关子查询
- 书写相关子查询
- 使用子查询更新和删除数据
- 使用 **EXISTS** 和 **NOT EXISTS** 操作符
- 使用 **WITH** 子句

# 子查询

子查询是嵌套在 **SQL** 语句中的另一个**SELECT** 语句




# 子查询

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

- 子查询 (内查询) 在主查询执行之前执行
- 主查询使用子查询的结果 (外查询)

# 子查询应用举例

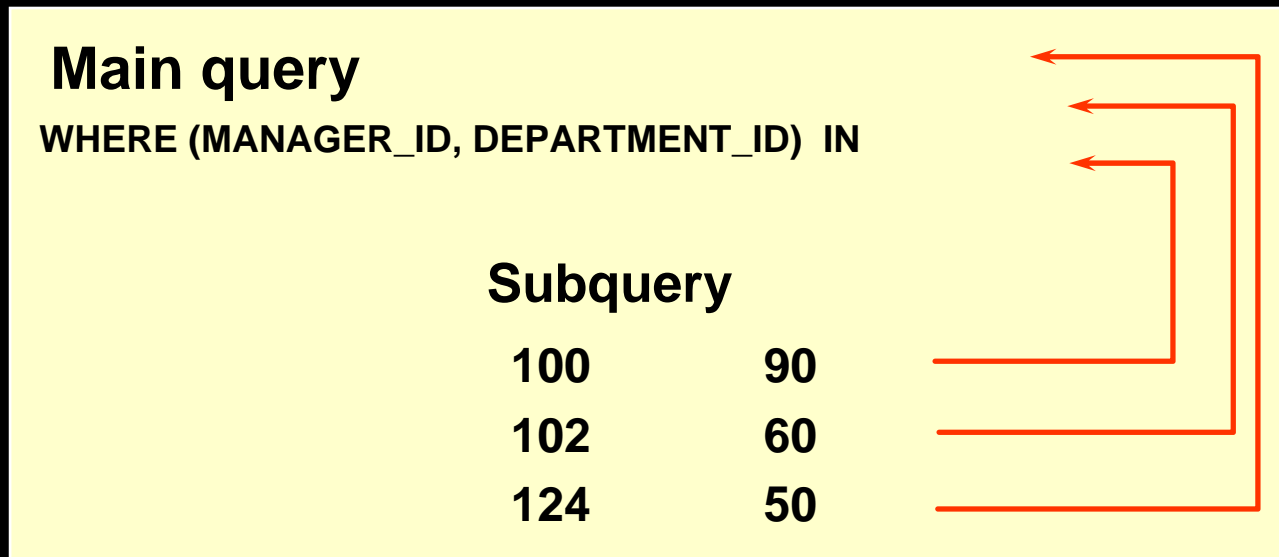
```
SELECT last_name
FROM employees
WHERE salary >
    (SELECT salary
     FROM employees
     WHERE employee_id = 149) ;
```



LAST_NAME
King
Kochhar
De Haan
Abel
Hartstein
Higgins

6 rows selected.

# 多列子查询



主查询与子查询返回的多个列进行比较



# 列比较

多列子查询中的比较分为两种:

- 成对比较
- 不成对比较

## 成对比较举例

```
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);
```

# 不成对比较举例

```
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);
```

# 在 FROM 子句中使用子查询

```
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

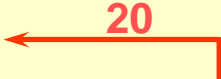
7 rows selected.

# 单列子查询表达式

- 单列子查询表达式是在一行中只返回一列的子查询
- **Oracle8i** 只在下列情况下可以使用, 例如:
  - **SELECT** 语句 (**FROM** 和 **WHERE** 子句)
  - **INSERT** 语句中的**VALUES**列表中
- **Oracle9i**中单列子查询表达式可在下列情况下使用:
  - **DECODE** 和 **CASE**
  - **SELECT** 中除 **GROUP BY** 子句以外的所有子句中

# 单列子查询应用举例

## 在 CASE 表达式中使用单列子查询

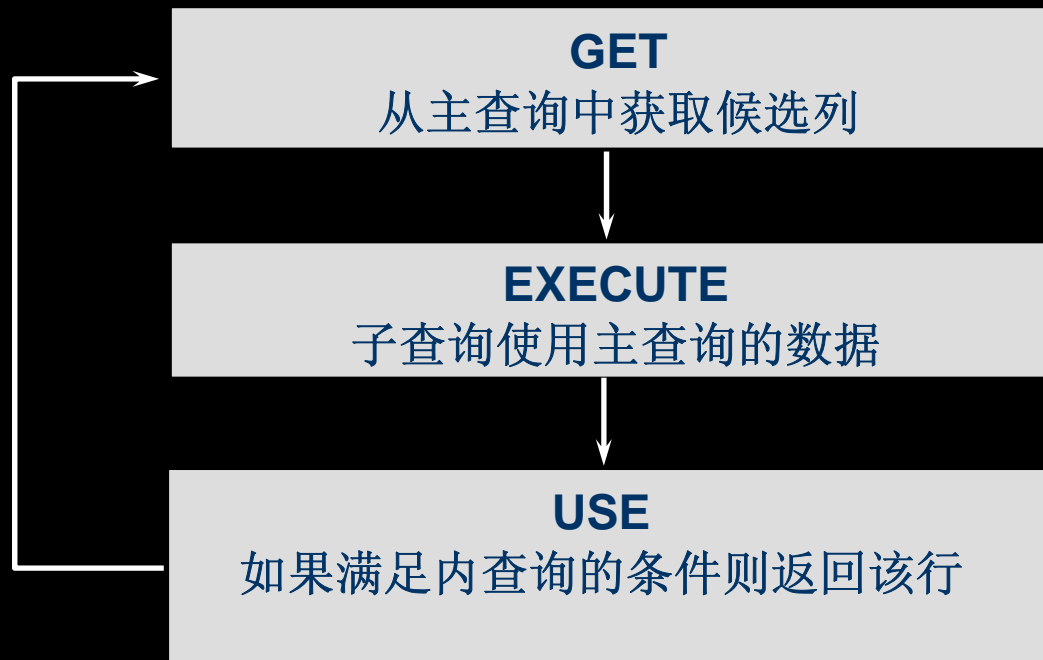
```
SELECT employee_id, last_name,  
       (CASE  
         WHEN department_id = 20  
          (SELECT department_id FROM departments  
           WHERE location_id = 1800)  
         THEN 'Canada' ELSE 'USA' END) location  
FROM   employees;
```

## 在 ORDER BY 子句中使用单列子查询

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

# 相关子查询

相关子查询按照一行接一行的顺序执行，主查询的每一行都执行一次子查询



# 相关子查询

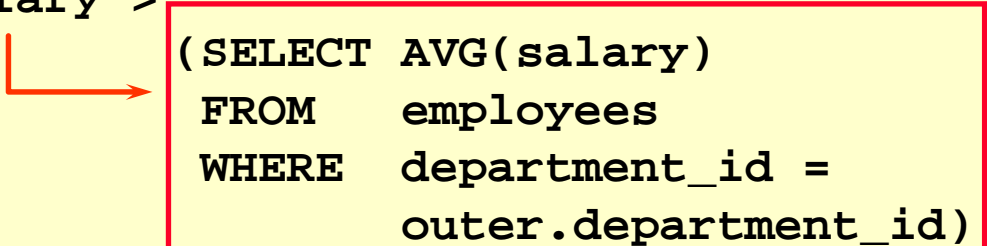
```
SELECT column1, column2, ...  
FROM   table1 outer  
WHERE  column1 operator  
        (SELECT column1, column2  
         FROM   table2  
         WHERE  expr1 =  
                outer.expr2);
```

子查询中使用主查询中的列



# 相关子查询举例

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



# 相关子查询举例

```
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

# EXISTS 操作符

- **EXISTS** 操作符检查在子查询中是否存在满足条件的行
- 如果在子查询中存在满足条件的行:
  - 不在子查询中继续查找
  - 条件返回 **TRUE**
- 如果在子查询中不存在满足条件的行:
  - 条件返回 **FALSE**
  - 继续在子查询中查找

# EXISTS 操作符应用举例

```
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.

# NOT EXISTS 操作符应用举例

```
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

# 相关更新

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

使用相关子查询依据一个表中的数据更新另一个表的数据

# 相关更新应用举例

```
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);
```

# 相关删除

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

使用相关子查询依据一个表中的数据删除另一个表的数据



# 相关删除应用举例

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

# WITH 子句

- 使用 **WITH** 子句, 可以避免在 **SELECT** 语句中重复书写相同的语句块
- **WITH** 子句将该子句中的语句块执行一次 并存储到用户的临时表空间中
- 使用 **WITH** 子句可以提高查询效率

# WITH 子句应用举例

**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;
```

# 总结

通过本章学习,您已经可以:

- 使用多列子查询
- 多列子查询的成对和非成对比较
- 单列子查询
- 相关子查询
- **EXISTS** 和 **NOT EXISTS**操作符
- 相关更新和相关删除
- **WITH**子句

# 19

## 分级查询

# 目标

通过本章学习，您将可以：

- 分级查询的概念
- 创建树形的报表
- 格式划分级数据
- 在树形结构中删除分支和节点

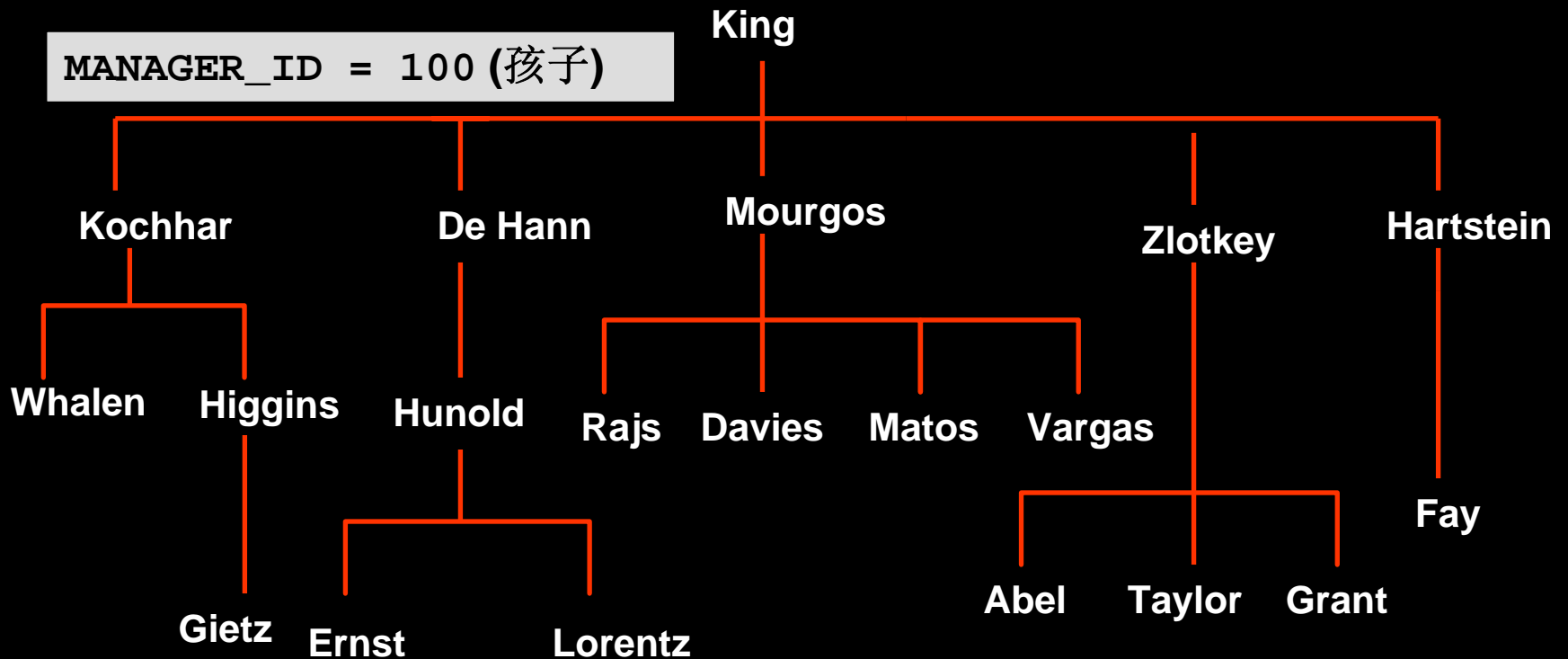
# EMPLOYEES 表中的数据

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.

# 树形结构

EMPLOYEE\_ID = 100 (双亲)





# 分级查询

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

**WHERE 条件:**

```
expr comparison_operator expr
```

# 遍历树

## 始点

- 指定需要满足的条件
- 接受有效的条件

```
START WITH column1 = value
```

遍历 **EMPLOYEES** 表, 以姓名为 **Kochhar**的员工作为始点

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

# 遍历树

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

从顶到底遍历 **EMPLOYEES** 表

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

方向

从顶到底



**Column1 = Parent Key**  
**Column2 = Child Key**

从底到顶



**Column1 = Child Key**  
**Column2 = Parent Key**

# 遍历树: 从底到顶

```
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	

# 遍历树: 从顶到底

```
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

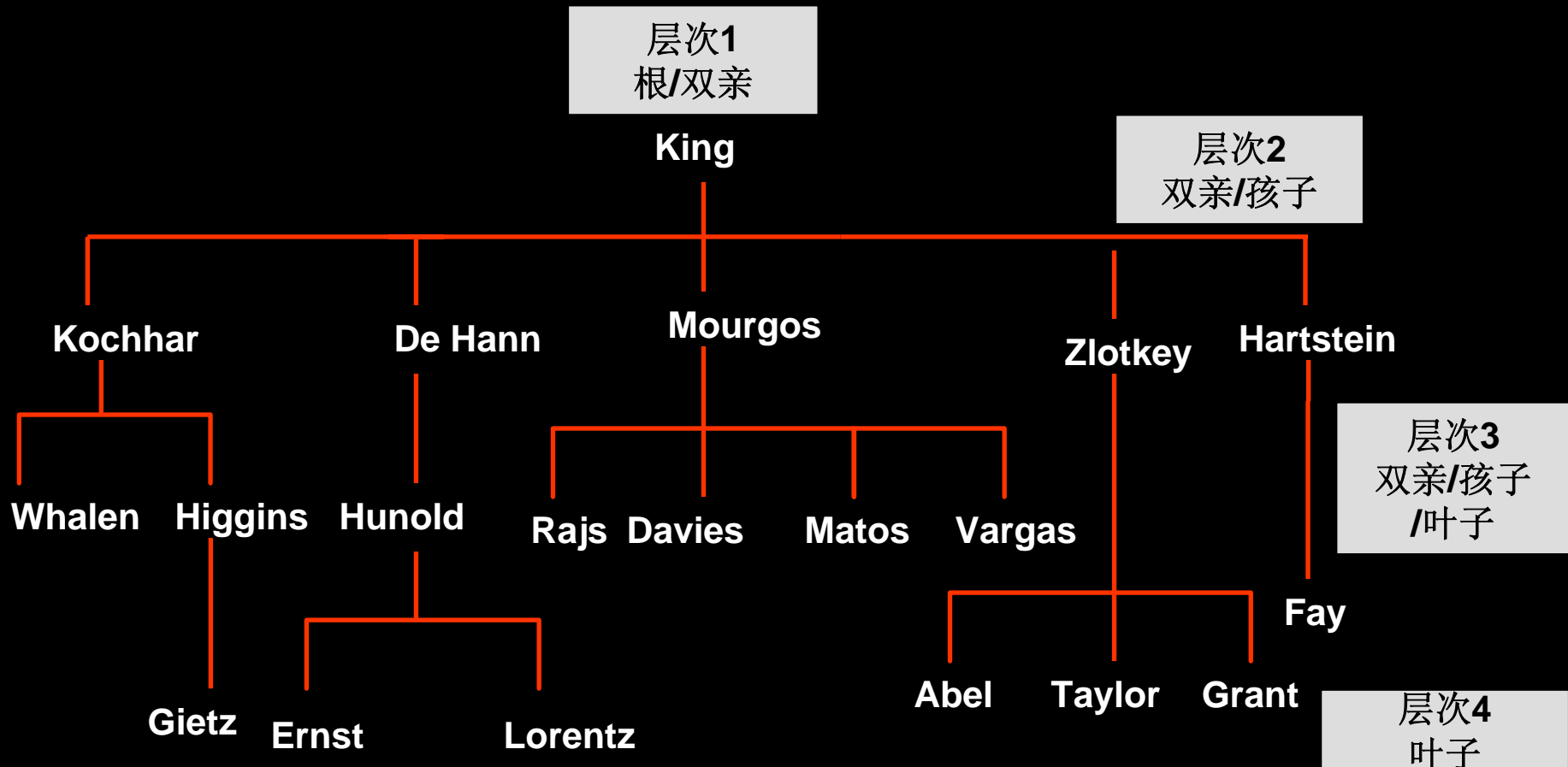
Grant reports to Zlotkey

Hartstein reports to King

Fay reports to Hartstein

20 rows selected.

# 使用 **LEVEL** 伪列标记层次



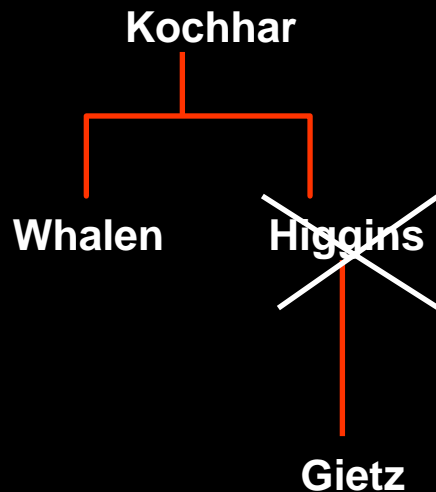
# 使用 LEVEL 和 LPAD格式化分层查询

```
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
```

# 修剪树枝

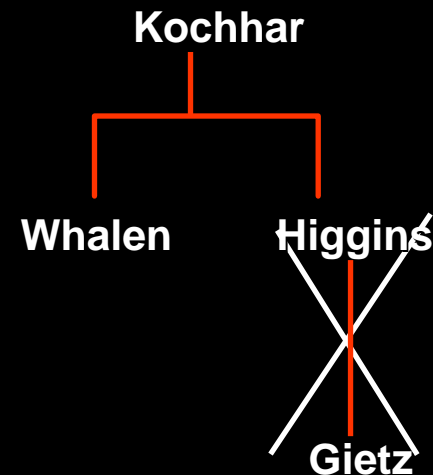
使用 **WHERE** 子句删除节点

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



使用 **CONNECT BY** 子句删除树枝

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





# 总结

通过本章学习,您已经可以:

- 对具有层次关系的数据创建树形报表
- 指定遍历的始点和方向
- 删除节点和树枝

# Oracle9i 对 DML 和 DDL 语句的扩展

# 目标

通过本章学习，您将可以：

- 描述多表插入的特点
- 使用不同类型的多表插入
  - 无条件的 **INSERT**
  - 旋转 **INSERT**
  - 有条件的 **ALL INSERT**
  - 有条件的 **FIRST INSERT**
- 创建和使用外部表
- 创建主键约束的同时创建索引

# INSERT 语句

- 使用 **INSERT** 语句向表中插入新的数据

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

- 使用上面的语句每次只能向表中插入一行数据

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

# UPDATE 语句

- 使用UPDATE 语句更新表中的数据

```
UPDATE          table
SET             column = value [, column = value, ...]
[WHERE         condition];
```

- 使用上面的语句每次可更新表中的一行或多行数据
- 使用 **WHERE** 子句指定更新的条件

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

# 多表 **INSERT** 语句

- **INSERT...SELECT** 是使用一个**DML** 语句向多个表中插入数据的一部分
- 多表**INSERT** 语句可作为数据仓库应用中向目标数据库传送数据的一种方法
- 它具有更高的效率：
  - 避免使用多各**DML** 语句
  - 使用一个**DML** 完成 **IF...THEN** 的逻辑处理

# 多表 **INSERT** 语句的类型

**Oracle9i** 提供以下四种多表**INSERT** 语句类型:

- 无条件的 **INSERT**
- 有条件的 **ALL INSERT**
- 有条件的 **FIRST INSERT**
- 旋转 **INSERT**

# 多表 INSERT 语句

## 语法

```
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]
```



# 无条件的 INSERT ALL 应用举例

```
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.
```

# 有条件的 INSERT ALL 应用举例

```
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.
```

# 有条件的 FIRST INSERT 应用举例

```
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.
```

# 旋转 INSERT 应用举例

```
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.

# 外部表

- 外部表是只读的表，其数据存储在数据库外的平面文件中
- 外部表的各种参数在 **CREATE TABLE** 语句中指定
- 使用外部表，数据可以存储到外部文件或从外部文件中上载数据到数据库
- 数据可以使用 **SQL** 访问，但不能使用 **DML** 后在外部表上创建索引

# 创建路径

创建外部表之前应先使用CREATE DIRECTORY语句创建路径

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

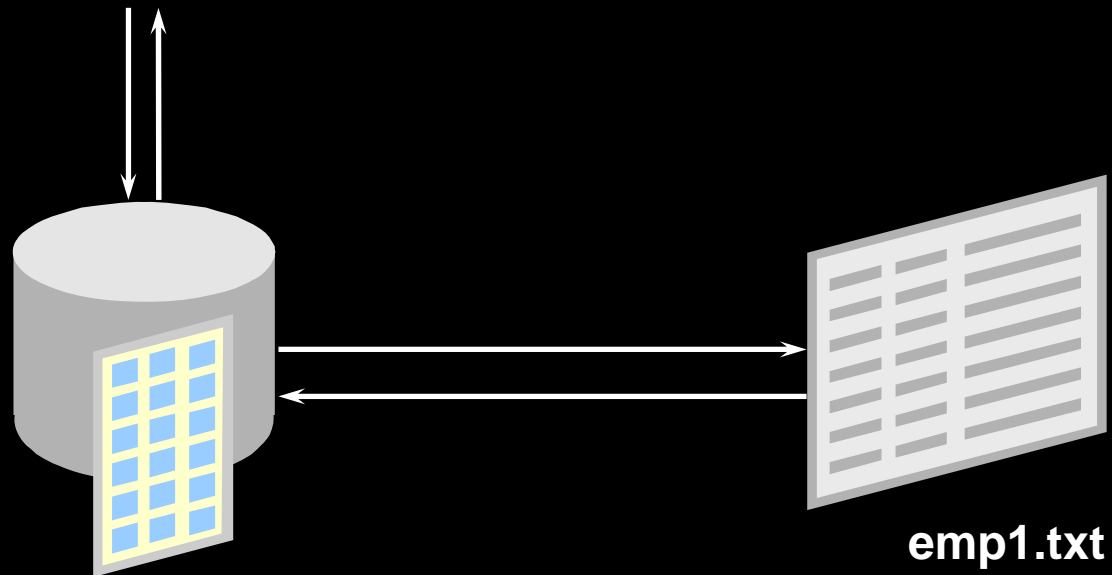
# 创建外部表举例

```
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 ('emp1.txt'))  
PARALLEL 5  
REJECT LIMIT 200;
```

Table created.

# 查询外部表

```
SELECT *  
FROM oldemp
```





# 创建主键约束同时创建索引举例

```
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

# 总结

通过本章学习,您已经可以:

- 使用多表插入代替多个单独的 **DML** 语句
- 创建外部表
- 创建主键约束同时创建索引