



# 基本 SQL SELECT 语句

讲师：佟刚

新浪微博：尚硅谷-佟刚

# 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。

# 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.

# 注意

- 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。

# 算术运算符

数字和日期使用的算术运算符。

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**’，别名使用**双引号**，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。

Xxx's email is xxx

# 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

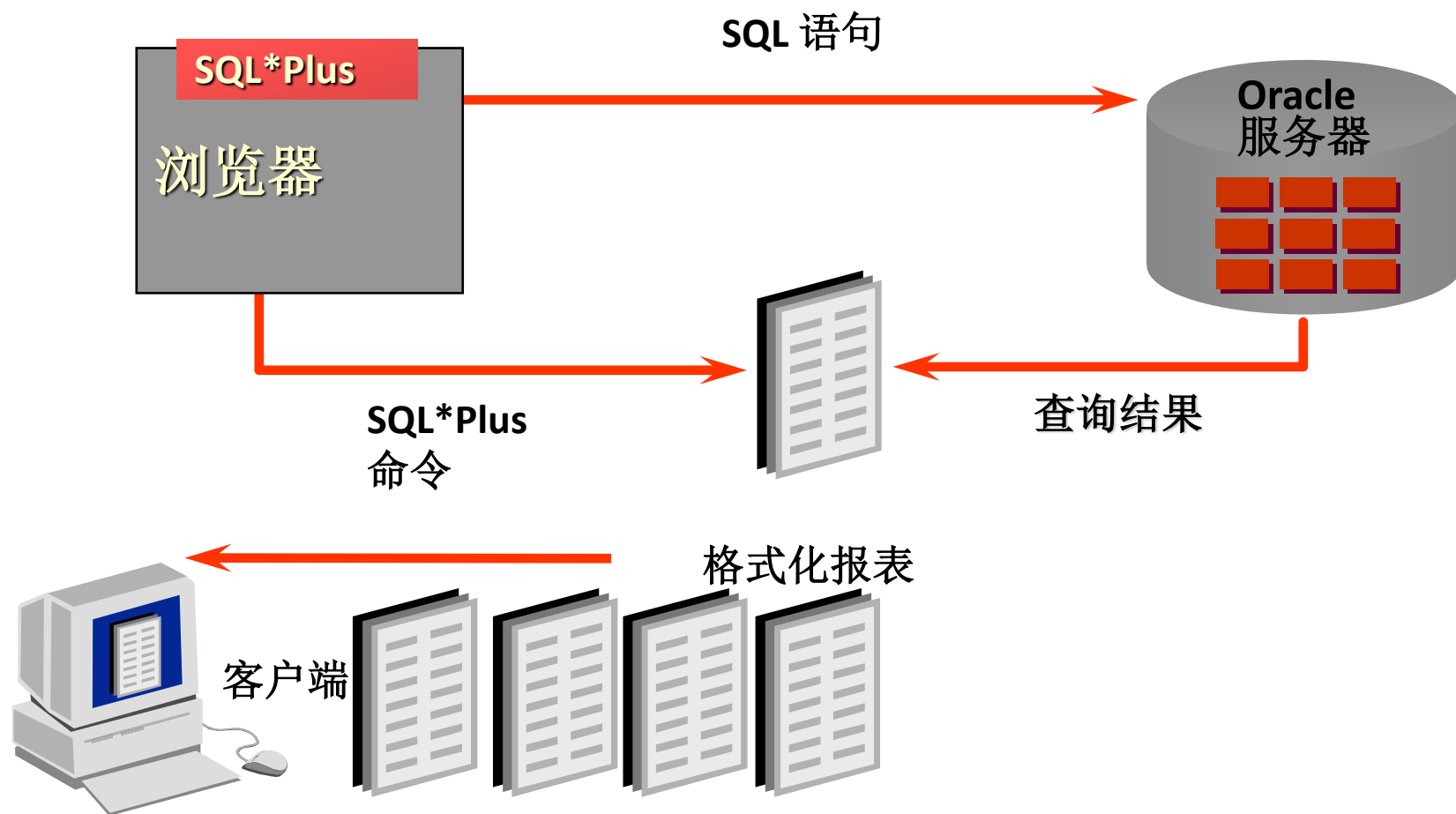
在 **SELECT** 子句中使用关键字 '**DISTINCT**' 删除重复行。

```
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\*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

Structural query language

# 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;
```







# 过滤和排序数据

讲师：佟刚

新浪微博：尚硅谷-佟刚

# 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号部门工作的  
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 (不是 ==)
>	大于
>=	大于、等于
<	小于
<=	小于、等于
<>	不等于 (也可以是 !=)

赋值使用 := 符号

Id = :id

# 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：
  - % 代表零个或多个字符(任意个字符)。
  - \_ 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。

# ESCAPE

- 回避特殊符号的：**使用转义符**。例如：将[%]转为[\%]、[\_]转为[\\_]，然后再加上[ESCAPE '\'] 即可。

```
SELECT job_id  
FROM jobs  
WHERE job_id LIKE 'IT\_%' escape '\\';
```

# NULL

使用 IS (NOT) 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
AND	逻辑并
OR	逻辑或
NOT	逻辑否



# AND

AND 要求并的关系为真。

```
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



# ORDER BY子句

- 使用 ORDER BY 子句排序
  - **ASC**(ascend): 升序
  - **DESC**(descend): 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

# 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 AND**, **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]];
```





# 单行函数

讲师：佟刚

新浪微博：尚硅谷-佟刚



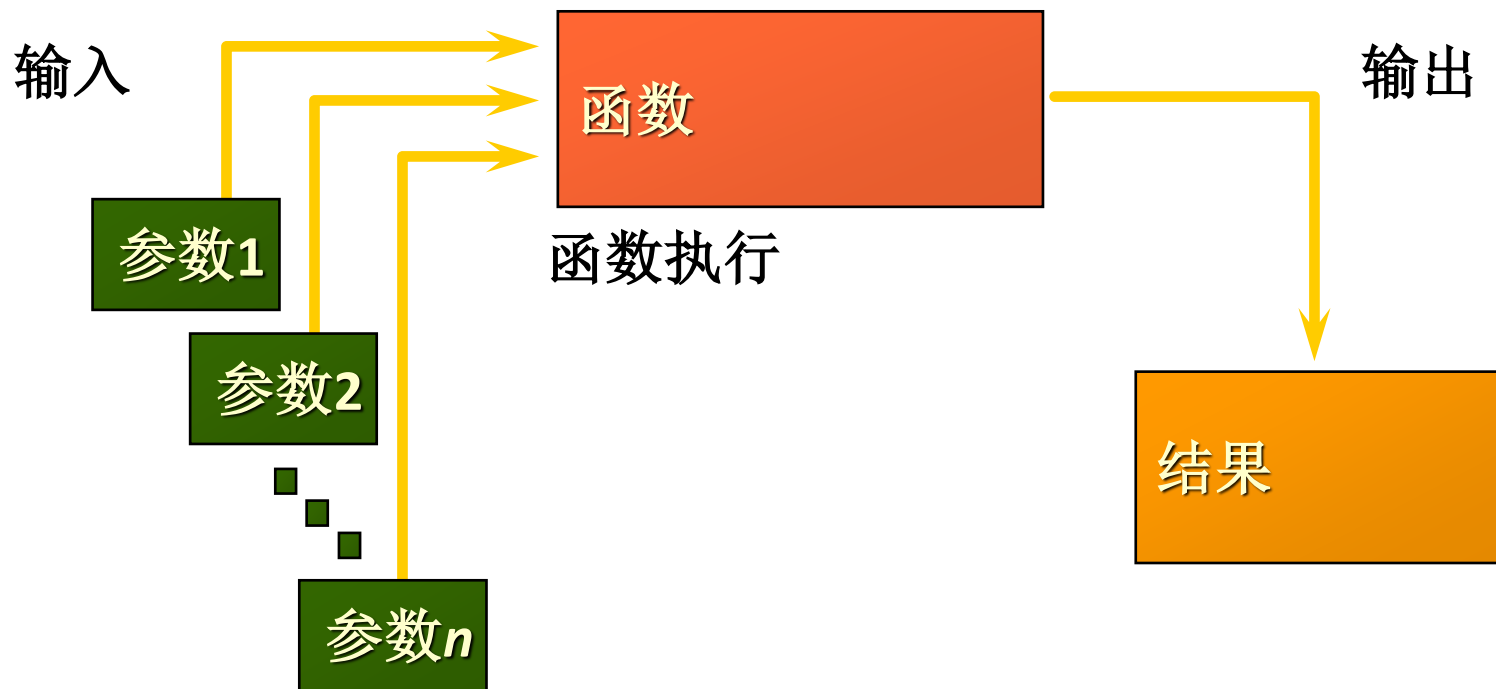
# 目标

通过本章学习，您将可以：

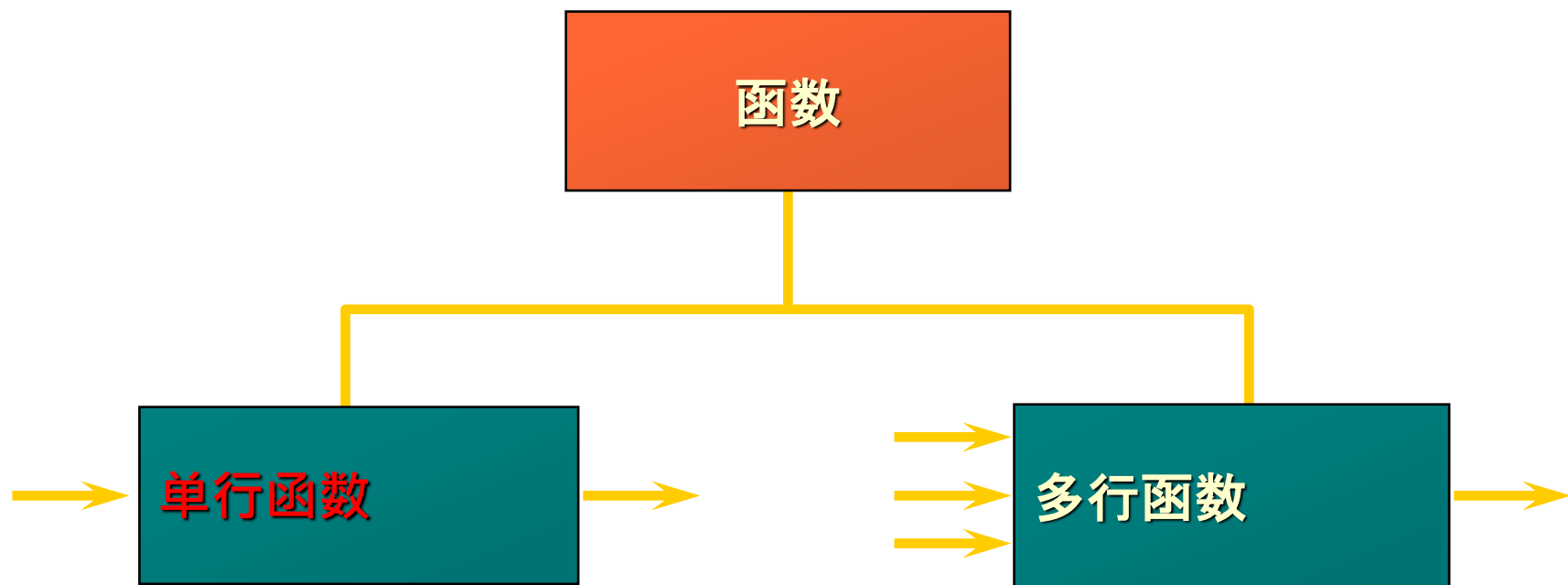
- SQL中不同类型的函数。
- 在 `SELECT` 语句中使用字符，数字和日期函数。
- 描述转换型函数的用途。



# SQL 函数



# 两种 SQL 函数



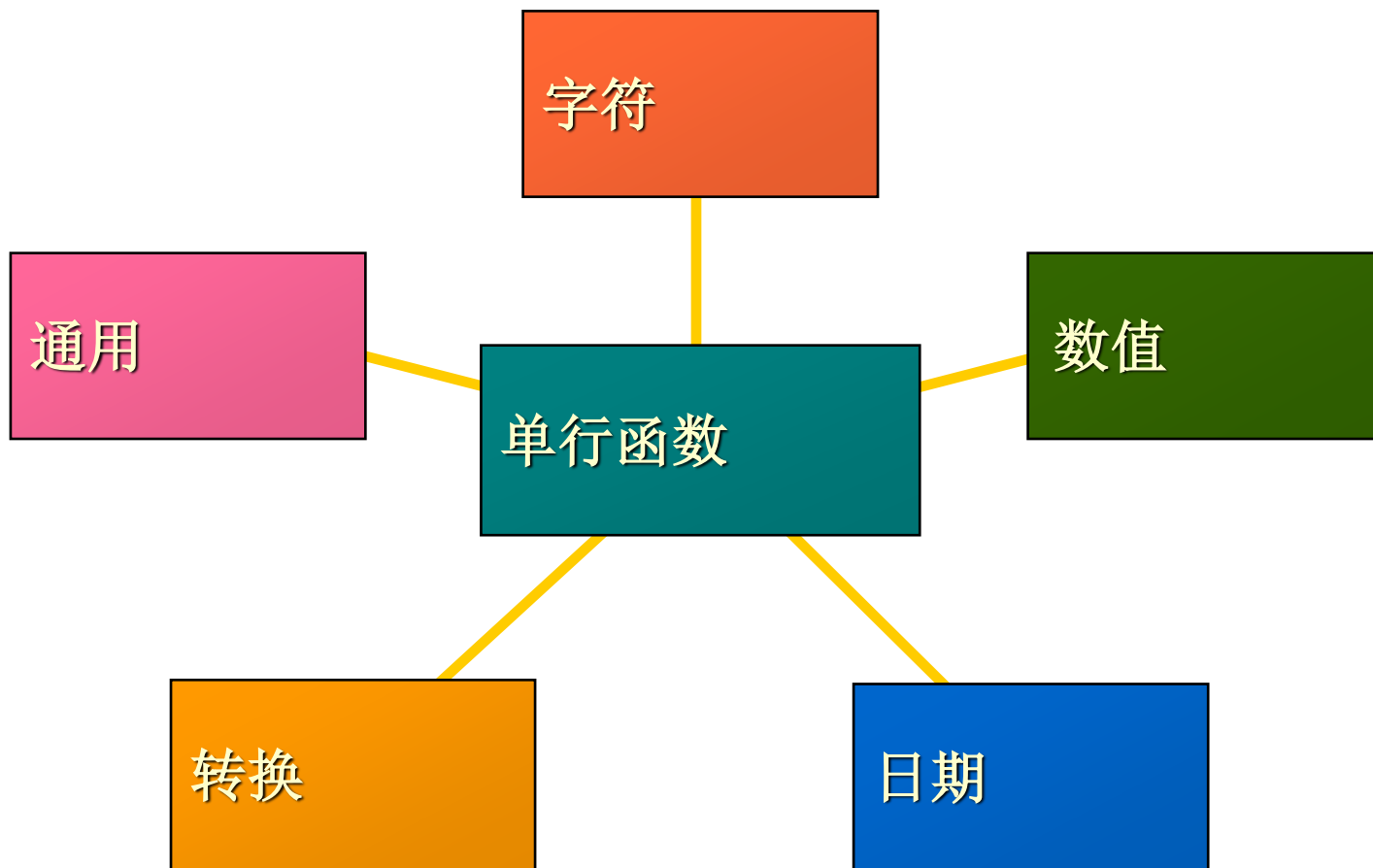
# 单行函数

单行函数:

- 操作数据对象
- 接受参数返回一个结果
- 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

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
<b>LOWER</b> ('SQL Course')	sql course
<b>UPPER</b> ('SQL Course')	SQL COURSE
<b>INITCAP</b> ('SQL Course')	Sql Course

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



# 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

# 数字函数

- ROUND: 四舍五入

ROUND (45.926, 2)      →      45.93

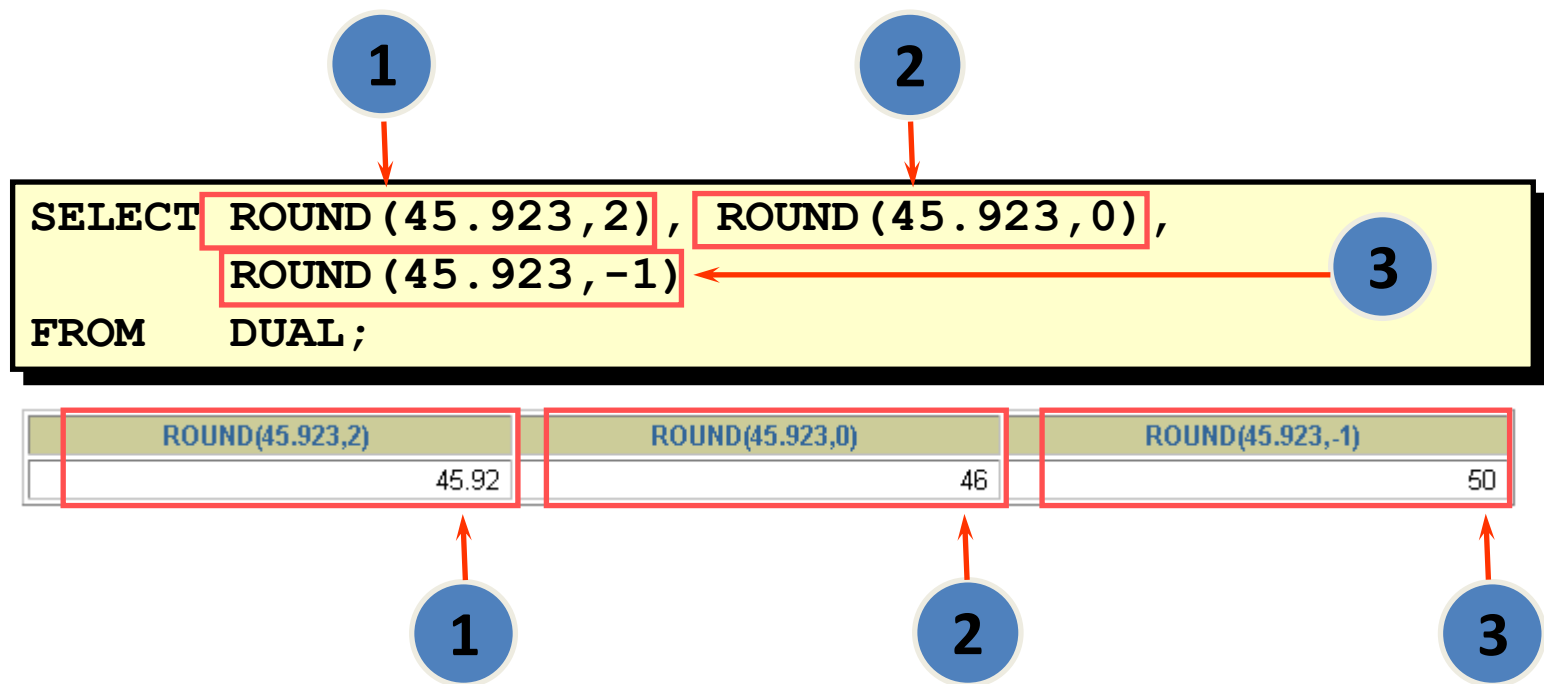
- TRUNC: 截断

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

1                      2

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

3

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

1                      2                      3



# 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 中的日期型数据实际含有两个值: 日期和时间。
- 默认的时间格式是 DD-MON-RR.

```
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'

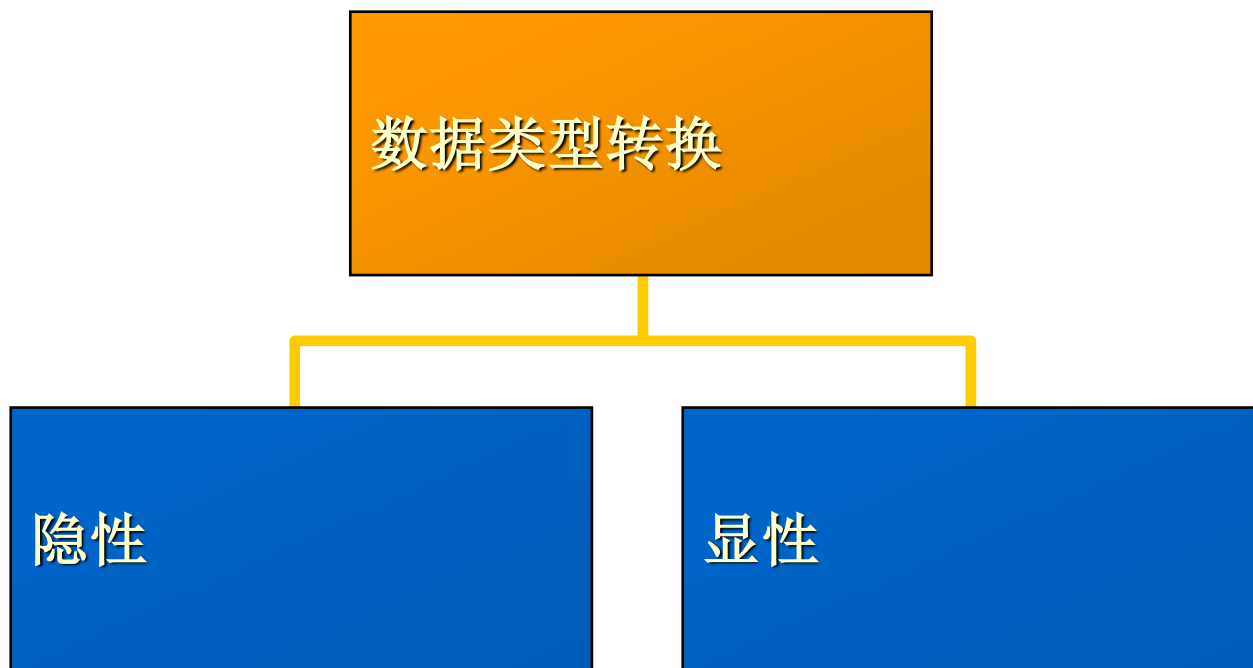
# 日期函数

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</b> or CHAR	<b>NUMBER</b>
VARCHAR2 or CHAR	<b>DATE</b>
NUMBER	VARCHAR2
DATE	VARCHAR2

## 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。
- 日期之间用逗号隔开。

# 日期格式的元素

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



# 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
---------------	---------------

# TO\_CHAR 函数对日期的转换

```
SELECT last_name,  
       TO_CHAR(hire_date, 'DD 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'])
```

- 使用 **TO\_DATE** 函数将字符转换成日期:

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

# 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



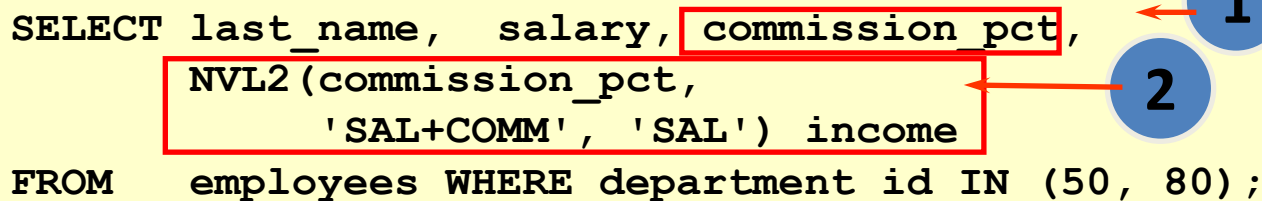
nvl(expr1, expr3)

nvl2(expr1, expr1, expr3)

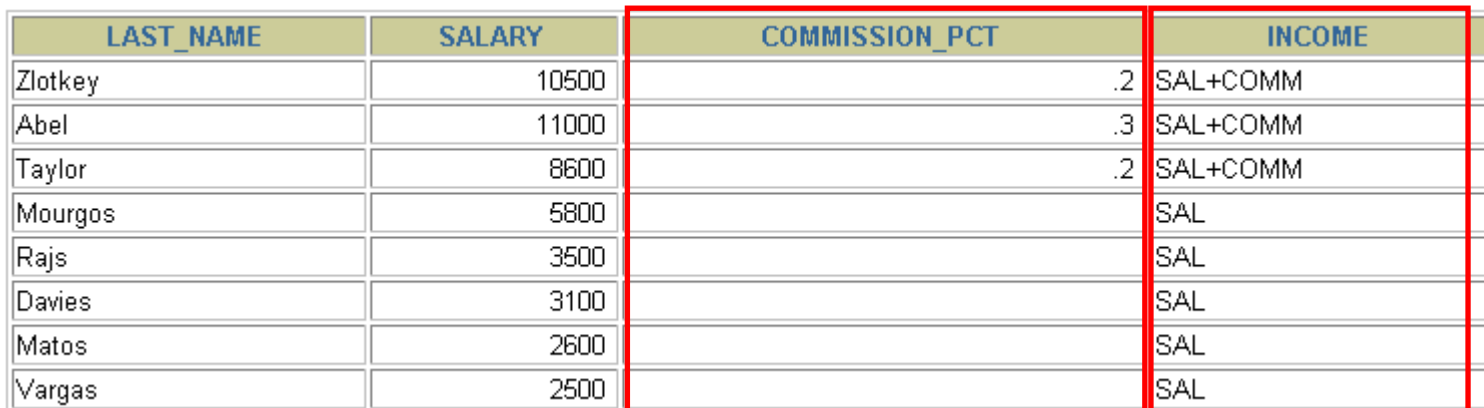
# 使用 NVL2 函数

**NVL2 (expr1, expr2, expr3) :** expr1不为NULL, 返回expr2; 为NULL, 返回expr3。

```
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

NULLIF (expr1, expr2) : 相等返回NULL,

不等返回expr1

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

FIRST_NAME	expr1	LAST_NAME	expr2	RESULT
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.

# 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,则返回下一个表达式, 对其其他的参数进行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;
```

# 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



# 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





# 多表查询

讲师：佟刚

新浪微博：尚硅谷-佟刚



# 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

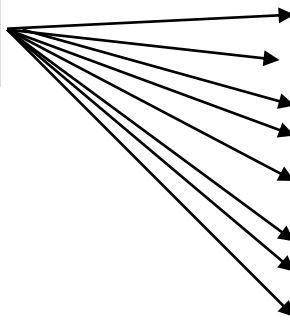
DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
60	IT	1400
80	Sales	2500
90	Executive	1700
110	Accounting	1700
190	Contracting	1700



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

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

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90



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

- **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.

...



# 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** 子句中写入连接条件。
- 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
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。
- 在不同表中具有相同列名的列可以用表的别名加以区分。

# 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

# 内连接和外连接(1)

- 内连接: 合并具有同一列的两个以上的表的行, **结果集中不包含一个表与另一个表不匹配的行**
- 外连接: 两个表在连接过程中除了返回满足连接条件的行以外**还返回左（或右）表中不满足条件的行，这种连接称为左（或右）外联接**。没有匹配的行时, 结果表中相应的列为空(NULL). 外连接的 WHERE 子句条件类似于内部链接, 但**连接条件中没有匹配行的表的列后面要加外连接运算符, 即用圆括号括起来的加号(+)**.

# 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**EMPLOYEE\_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.



## 内连接和外连接(2)

- 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.







# 分组函数

讲师：佟刚

新浪微博：尚硅谷-佟刚

# 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 子句数据分组。
- 使用HAVING 子句过滤分组结果集。



# 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

4400

9500

求出

**EMPLOYEES**

表中各

部门的

平均工资

6400

10033

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

...

20 rows selected.

# 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.

部门最高工资  
比 10000 高的  
部门

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



# 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] ;
```







# 子查询

讲师：佟刚

新浪微博：尚硅谷-佟刚



# 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。
- 列出子查询的类型。
- 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

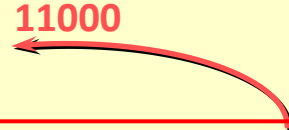
```
SELECT    select_list  
FROM      table  
WHERE     expr operator
```

```
(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。
- 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

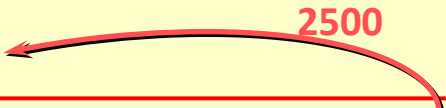
```
SELECT last_name, job_id, salary
FROM employees
WHERE job_id = ST_CLERK
AND salary > 2600
      (SELECT job_id
       FROM employees
       WHERE employee_id = 141)
      (SELECT salary
       FROM employees
       WHERE employee_id = 143);
```

LAST_NAME	JOB_ID	SALARY
Rajs	ST_CLERK	3500
Davies	ST_CLERK	3100



# 在子查询中使用组函数

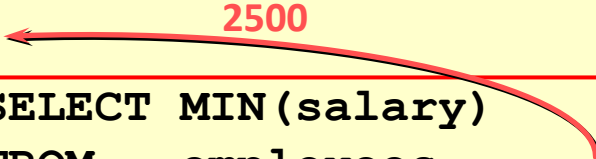
```
SELECT last_name, job_id, salary
FROM employees
WHERE salary =  
      (SELECT MIN(salary)  
       FROM employees);
```



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：

- 如何使用子查询。
- 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);
```





# 处理数据

讲师：佟刚

新浪微博：尚硅谷-佟刚



# 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
- 控制事务



# 数据控制语言

- DML(Data Manipulation Language – 数据操作语言)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

# 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', 'Raphealy',
             '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	Raphealy	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与 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.



# 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(Data Definition Language – 数据定义语言) 语句
- 一个 DCL(Data Control Language – 数据控制语言) 语句

# 数据库事务

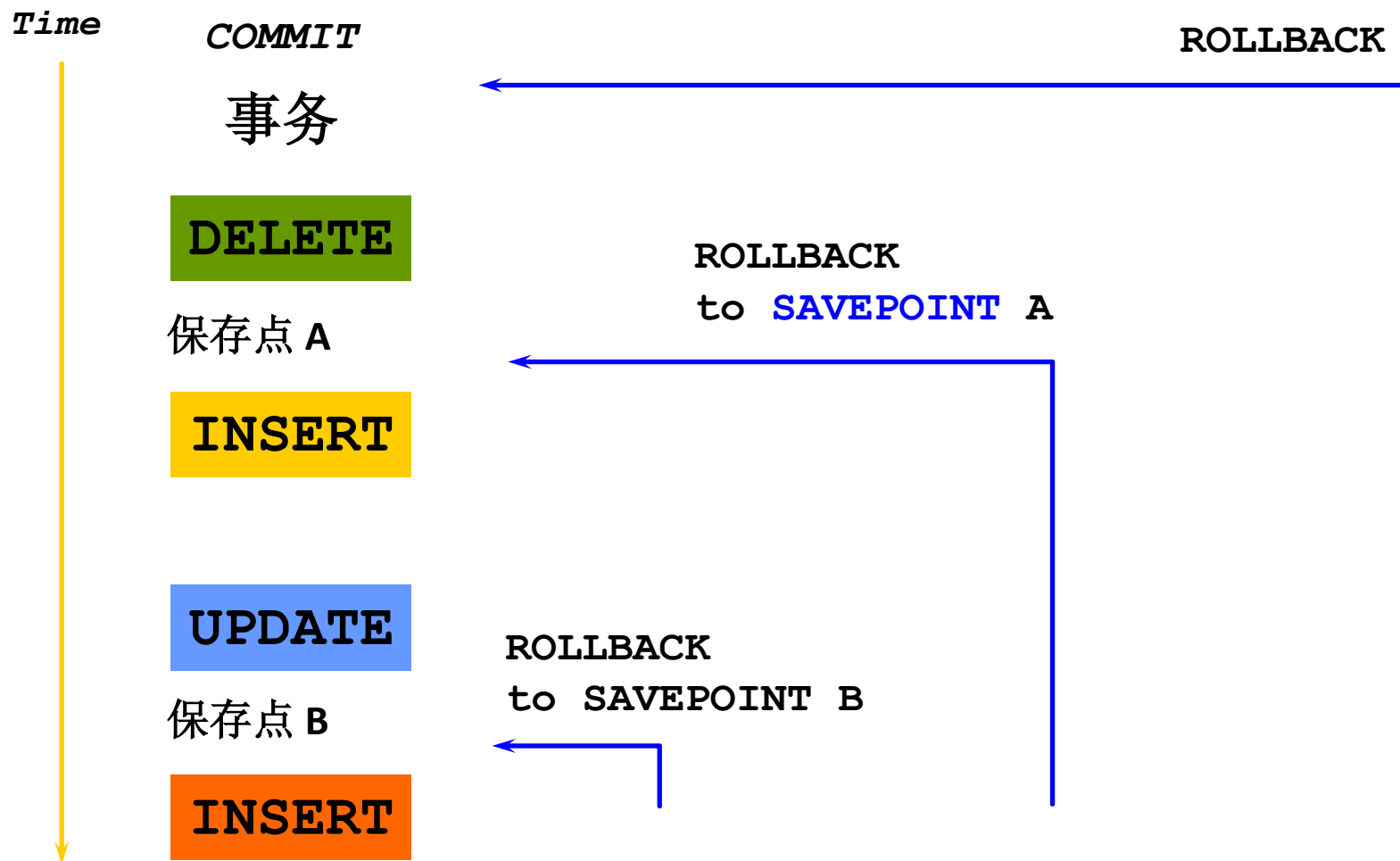
- 以第一个 DML 语句的执行作为开始
- 以下面的其中之一作为结束:
  - **COMMIT 或 ROLLBACK 语句**
  - DDL 或 DCL 语句（**自动提交**）
  - 用户会话正常结束
  - 系统异常終了

# 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。
- 修改前的数据状态被恢复。
- 锁被释放。

```
DELETE FROM copy_emp;
```

```
22 rows deleted.
```

```
ROLLBACK;
```

```
Rollback complete.
```

# 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
INSERT	插入
UPDATE	修正
DELETE	删除
COMMIT	提交
SAVEPOINT	保存点
ROLLBACK	回滚

# 数据库的隔离级别

- 对于同时运行的多个事务, 当这些事务访问数据库中相同的数据时, 如果没有采取必要的隔离机制, 就会导致各种并发问题:
  - **脏读**: 对于两个事物 T1, T2, T1 读取了已经被 T2 更新但还没有被提交的字段. 之后, 若 T2 回滚, T1 读取的内容就是临时且无效的.
  - **不可重复读**: 对于两个事物 T1, T2, T1 读取了一个字段, 然后 T2 更新了该字段. 之后, T1 再次读取同一个字段, 值就不同了.
  - **幻读**: 对于两个事物 T1, T2, T1 从一个表中读取了一个字段, 然后 T2 在该表中插入了一些新的行. 之后, 如果 T1 再次读取同一个表, 就会多出几行.
- 数据库事务的隔离性: 数据库系统必须具有隔离并发运行各个事务的能力, 使它们不会相互影响, 避免各种并发问题.
- 一个事务与其他事务隔离的程度称为隔离级别. 数据库规定了多种事务隔离级别, 不同隔离级别对应不同的干扰程度, 隔离级别越高, 数据一致性就越好, 但并发性越弱



# 数据库的隔离级别

- 数据库提供的 4 种事务隔离级别:

隔离级别	描述
READ UNCOMMITTED (读未提交数据)	允许事务读取未被其他事物提交的变更. 脏读, 不可重复读和幻读的问题都会出现
READ COMMITED (读已提交数据)	只允许事务读取已经被其它事务提交的变更. 可以避免脏读, 但不可重复读和幻读问题仍然可能出现
REPEATABLE READ (可重复读)	确保事务可以多次从一个字段中读取相同的值. 在这个事务持续期间, 禁止其他事物对这个字段进行更新. 可以避免脏读和不可重复读, 但幻读的问题仍然存在.
SERIALIZABLE(串行化)	确保事务可以从一个表中读取相同的行. 在这个事务持续期间, 禁止其他事务对该表执行插入, 更新和删除操作. 所有并发问题都可以避免, 但性能十分低下.

- Oracle 支持的 2 种事务隔离级别: **READ COMMITED**, **SERIALIZABLE**. Oracle 默认的事务隔离级别为: **READ COMMITED**
- Mysql 支持 4 中事务隔离级别. Mysql 默认的事务隔离级别为: **REPEATABLE READ**





# 创建和管理表

讲师：佟刚

新浪微博：尚硅谷-佟刚



# 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

# 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 ;
```

# 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

\$abc, **2abc**, \_abc, a-b, **a#d**



## 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

# 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)

# 数据类型

text

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

# 使用子查询创建表

- 使用 `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);
```

```
ALTER TABLE table_name rename column old_column_name  
to new_column_name
```

# 追加一个新列

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

新列

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.
```

# 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

# 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







# 约束

讲师：佟刚

新浪微博：尚硅谷-佟刚

# 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
- 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



# 表级约束和列级约束

- 作用范围：列级约束只能作用在一个列上，而表约束可以作用在多个列上（当然表约束也可以作用在一个列上）。
- 定义方式：列约束必须跟在列的定义里后面，表约束不与列一起，而是单独定义。
- **非空(not null) 约束只能定义在列上**

# 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

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

# 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时，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

```
..., 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.
```



# 无效化约束

- 在ALTER TABLE 语句中使用 DISABLE 子句将约束无效化。

```
ALTER TABLE          employees  
DISABLE CONSTRAINT    emp_emp_id_pk;  
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

# 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





# 视图

讲师：佟刚

新浪微博：尚硅谷-佟刚



# 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

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

EMPLOYEE_ID	LAST_NAME	SALARY	HIRE_DATE	JOB_ID	SALARY
149	Zlotkey	10500	29-JAN-00	SA_MAN	10500
174	Abel	11000	24-MAY-96	SA_REP	11000
176	Taylor	8600	24-MAR-98	SA_REP	8600
178	Kimberely	7000	24-MAY-99	SA_REP	7000
200	Jennifer	4400	17-SEP-87	AD_ASST	4400
201	Michael	13000	17-FEB-96	MK_MAN	13000
202	Pat	6000	17-AUG-97	MK_REP	6000
205	Shelley	12000	07-JUN-94	AC_MGR	12000
206	William	8300	07-JUN-94	AC_ACCOUNT	8300

# 视图

- 视图是一种虚表.
- **视图建立在已有表的基础上**, 视图赖以建立的这些表称为**基表**。
- 向视图提供数据内容的语句为 **SELECT** 语句, 可以将视图理解为**存储起来的 **SELECT** 语句**.
- 视图向用户提供基表数据的另一种表现形式

# 为什么使用视图

- 控制数据访问
- 简化查询
- 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
DML 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

## SQL\*Plus

```
SELECT *  
FROM empvu80;
```

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

## Oracle Server

### USER\_VIEWS

#### EMPVU80

```
SELECT employee_id,  
       last_name, salary  
FROM   employees  
WHERE  department_id=80;
```

EMPLOYEES

# 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

# 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.
```

# 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
- Top-N 分析







# 其他数据库对象

讲师：佟刚

新浪微博：尚硅谷-佟刚



# 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: 可供多个用户用来产生唯一数值的数据库对象

- 自动提供唯一的数值
- 共享对象
- **主要用于提供主键值**
- 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
- 如果不将序列的值装入内存(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

# 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:

- 一种独立于表的模式对象, 可以存储在与表不同的磁盘或表空间中
- 索引被删除或损坏, 不会对表产生影响, 其影响的只是**查询**的速度
- 索引一旦建立, Oracle 管理系统会对其进行自动维护, 而且由 Oracle 管理系统决定何时使用索引. 用户不用在查询语句中指定使用哪个索引
- 在删除一个表时, 所有基于该表的索引会自动被删除
- 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

# 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

# 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';
```

# 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
- 使用索引提高查询效率
- 为数据对象定义同义词







# 高级子查询

讲师：佟刚

新浪微博：尚硅谷-佟刚



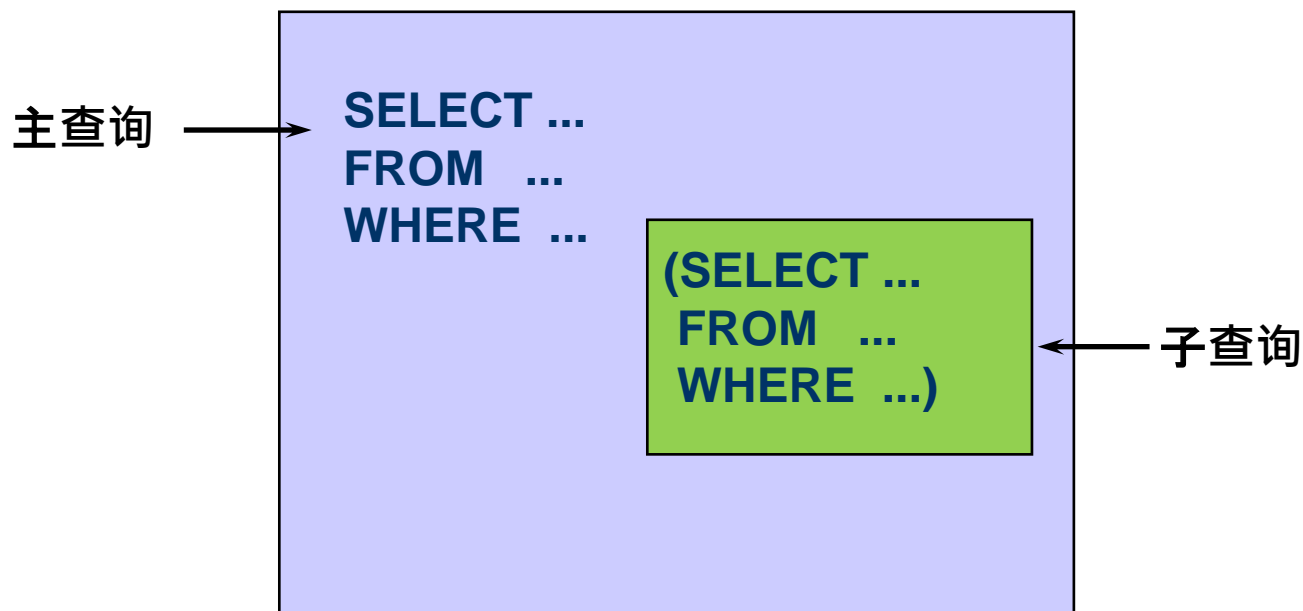
# 目标

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 > ← 10500  
(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

**Main query**

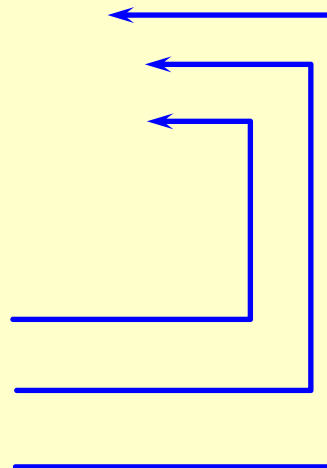
WHERE (MANAGER\_ID, DEPARTMENT\_ID) IN

**Subquery**

100          90

102          60

124          50



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

- 查询和 178, 174 在同一个部门，且同一个 leader 的员工有哪些？

```
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);
```

## 成对比较练习

- 查询和 191 号员工同一月入职，在同一个部门的员工的信息
- 查询和 105 号员工在同一个城市，且做同样工作的员工信息



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

## 不成对比较练习

- 查询和 166 号员工在同一个部门，且工作的城市在 Oxford 的员工的 last\_name, job\_id, salary
- 查询和 Fox 同年入职，且做 Sales Representative 工作的员工的 last\_name, job\_id, hire\_date

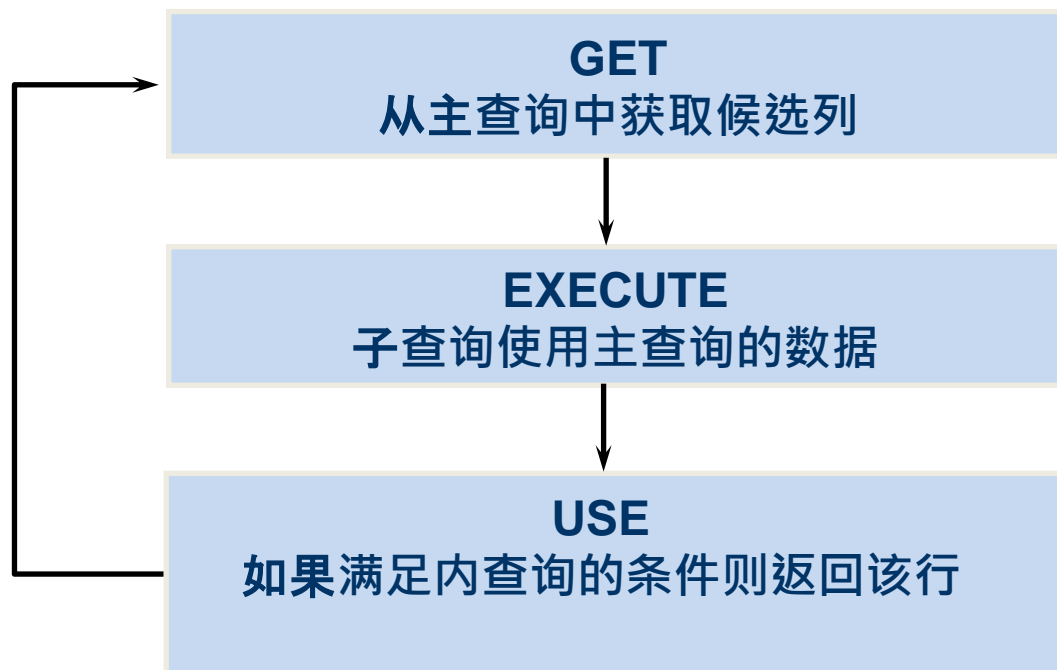
# 在 FROM 子句中使用子查询

查询工资比所在部门平均工资高的员工的  
last\_name, salary, department\_id, salary

```
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;
```

# 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) ;
```

# 相关子查询练习

1. 查询比做同样的工作的平均工资高的员工的  
employee\_id, last\_name, job\_id
2. 查询换过两次工作以上的员工的 employee\_id,  
last\_name, job\_id

# 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

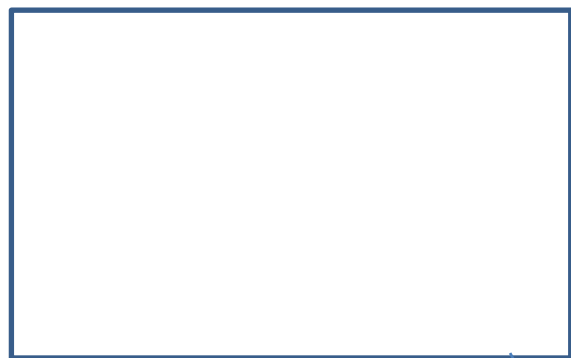
8 rows selected.

# 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

**IN**适合于外表大而内表小的情况；**EXISTS**适合于外表小而内表大的情况。



是否满足条件



是否存在



**IN**适合于外表大而内表小的情况；**EXISTS** 适合于外表小而内表大的情况。

# 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







# SET 运算符

讲师：佟刚

新浪微博：尚硅谷-佟刚

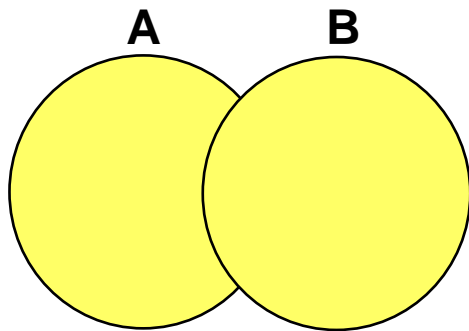


# 目标

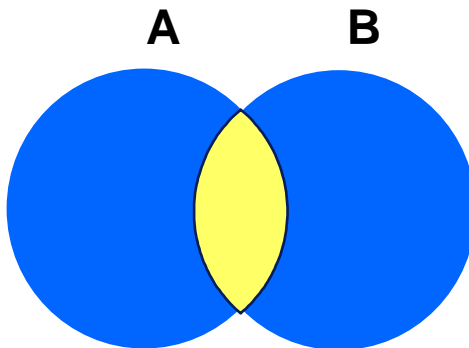
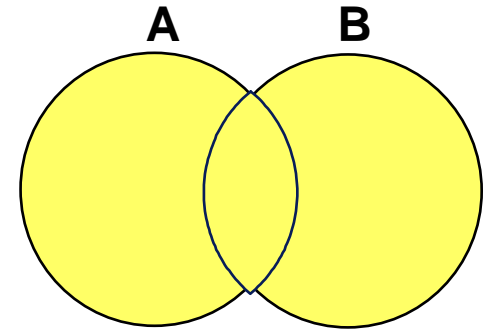
通过本章学习，您将可以：

- 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

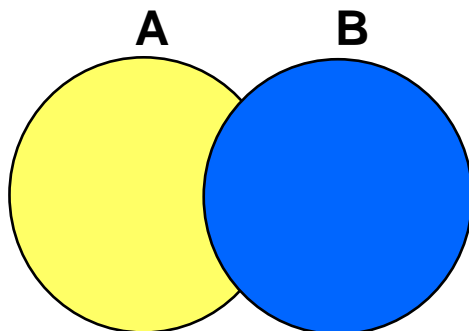
# SET 操作符



UNION/UNION ALL



INTERSECT

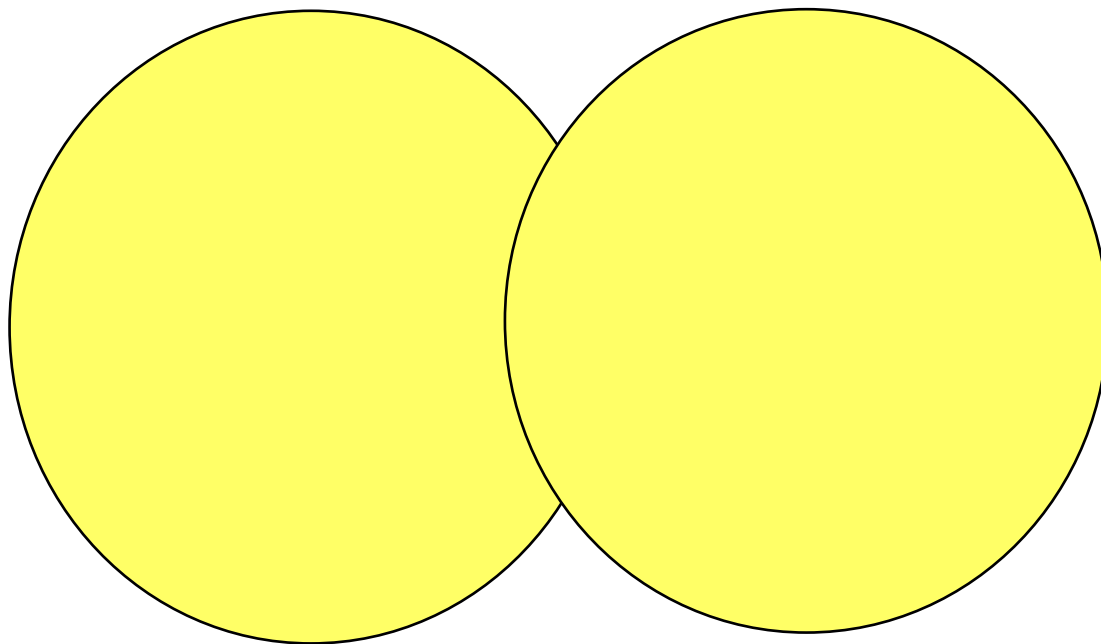


MINUS

# UNION 操作符

A

B



UNION 操作符返回两个查询的结果集的并集

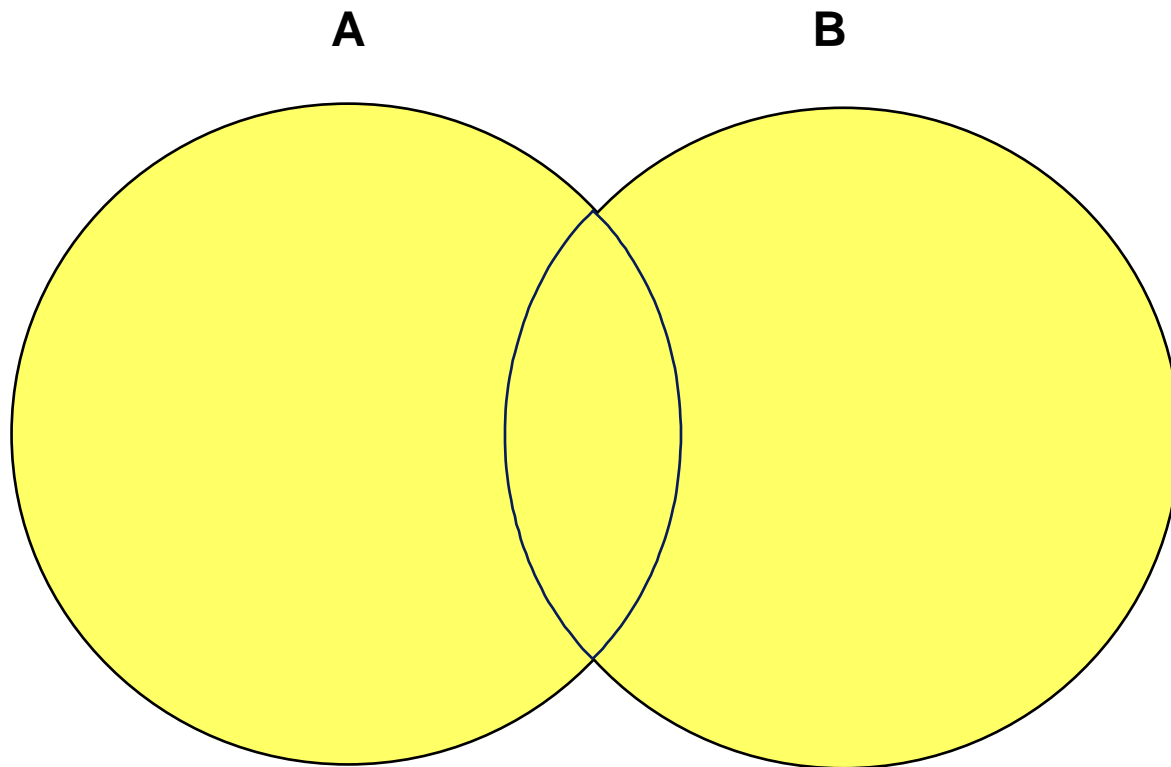
# UNION 操作符举例

```
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**)

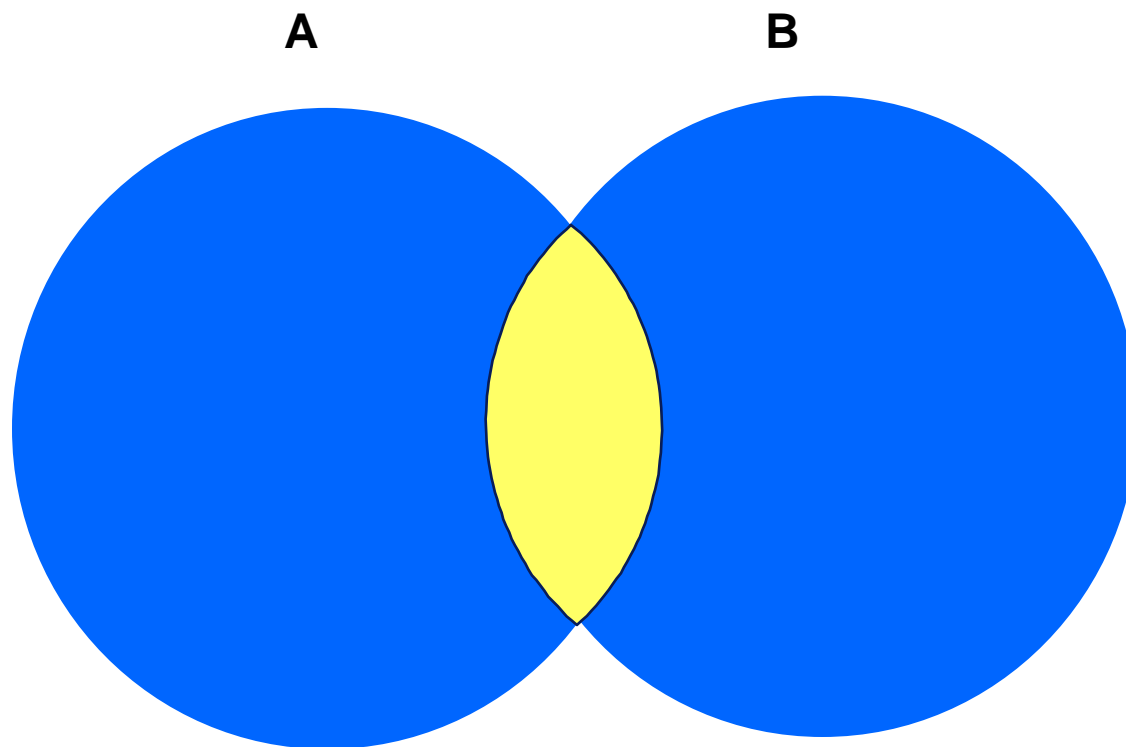
# UNION ALL 操作符举例

```
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

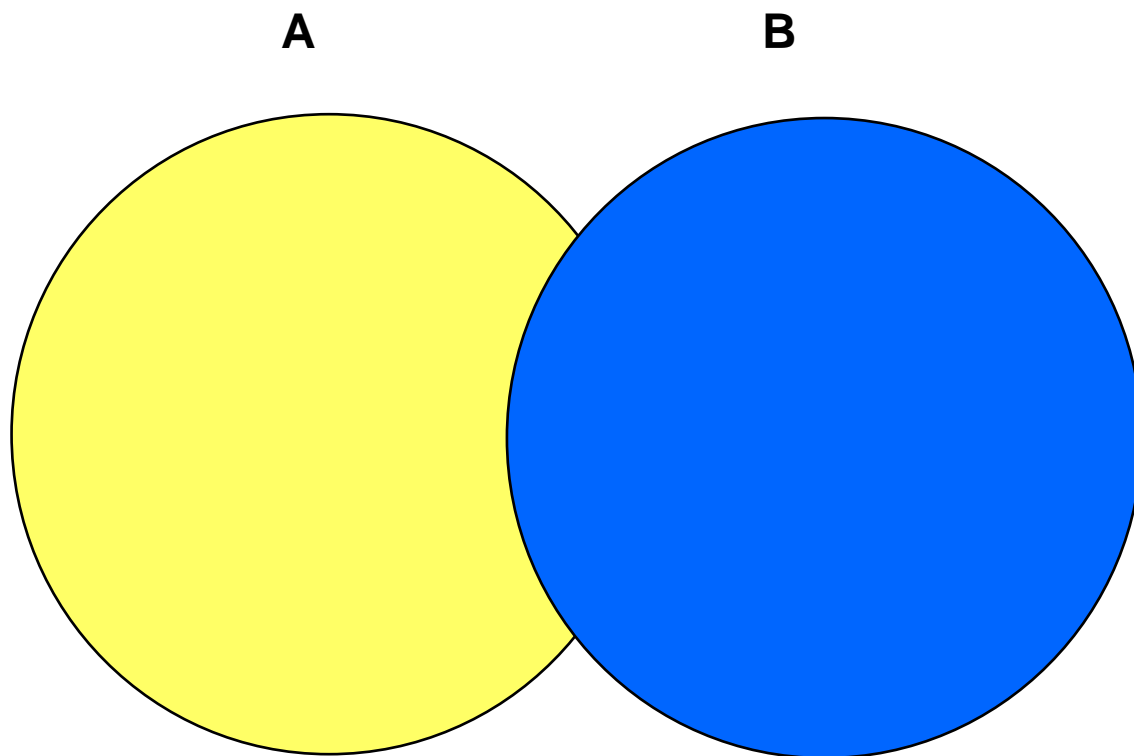
# INTERSECT 操作符举例

```
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或别名

# 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.

# 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

