

Oracle SQL培训

快速响应业务需求，提升业务运营及管理水平

L E A D E R

得帆

Oracle中间件技术服务领跑者

利用云计算技术搭建高效、稳定、安全的应用集成融合平台

USING CLOUD COMPUTING TECHNOLOGY TO BUILD A HIGHLY EFFICIENT, STABLE AND SECURE INTEGRATED APPLICATION INTEGRATION PLATFORM

- 灵动搭建应用
- 应用流程一体化
- 极速应用体验
- 应用管理便捷化



讲解人：徐翔轩



技术服务热线：

400-087-0500

得帆公司 版权所有

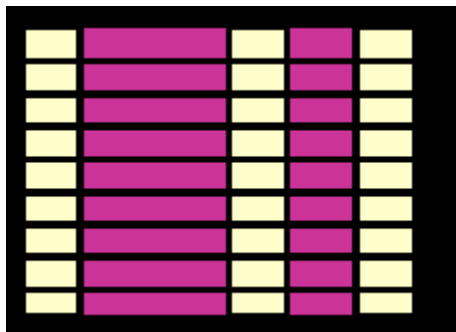


第一单元: SELECT 语句

SELECT 语句的作用:

列选择:

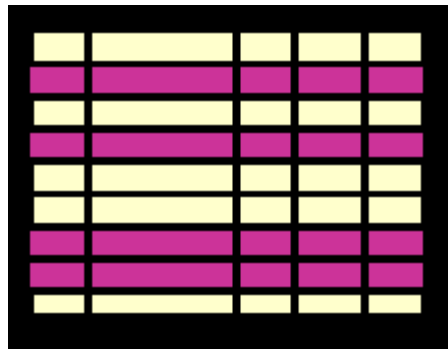
SELECT Column1, Column2 from table1



department_id

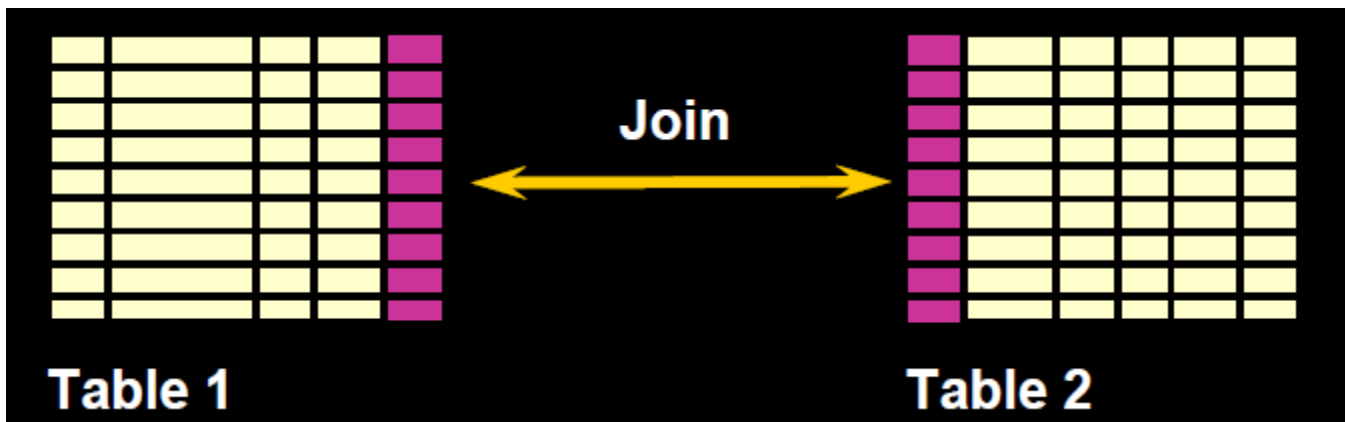
行选择:

SELECT * from table1



多表连接查询:

SELECT table1.Column1,
table2.Column2 from table1,table2...



第一单元：SELECT 语句

SELECT语句语法：

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

- SELECT 表示要取哪些列
- FROM 表示要从哪些表中取

SQL语句中的数学表达式：对于数值和日期型字段，可以进行 “加减乘除”

```
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 |

.....

第一单元：SELECT 语句

关于NULL的概念：

NULL表示 不可用、未赋值、不知道、不适用，它既不是0 也不是空格。

记住：一个数值与NULL进行四则运算，其结果是？ NULL

在Select的时候给列起个别名（注意双引号的作用）：

```
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 |

第一单元：SELECT 语句

字符串连接操作符：“||”

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

```
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 |

.....

第一单元：SELECT 语句

DISTINCT 去除重复行：

SELECT department_id FROM employees; 默认情况，返回所有行，包括重复行

| DEPARTMENT_ID | |
|---------------|----|
| | 90 |
| | 90 |
| | 90 |
| | 60 |
| | 60 |
| | 60 |
| | 50 |
| | 50 |
| | 50 |

SELECT DISTINCT department_id FROM employees; 使用DISTINCT消除重复结果行

| DEPARTMENT_ID | |
|---------------|-----|
| | 10 |
| | 20 |
| | 50 |
| | 60 |
| | 80 |
| | 90 |
| | 110 |
| | |

3 rows selected.

第二单元：条件限制和排序

条件限制的关键词：WHERE

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

比如：

| 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 |

```
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 |

第二单元：条件限制和排序

比较操作符：

| 比较操作符 | 意义 |
|-------------------|--------------------|
| = | 等于 |
| > | 大于 |
| >= | 大于等于 |
| < | 小于 |
| <= | 小于等于 |
| <> | 不等于 |
| BETWEEN ...AND... | 在两个值之间 |
| IN(set) | 在一个集合范围内 |
| LIKE | 匹配一个字符串样子，可以使用%通配符 |
| IS NULL | 是一个空值，注意不能使用 =NULL |

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

| LAST_NAME | SALARY |
|-----------|--------|
| Matos | 2600 |
| Vargas | 2500 |

第二单元：条件限制和排序

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

| LAST_NAME | SALARY |
|-----------|--------|
| Rajs | 3500 |
| Davies | 3100 |
| Matos | 2600 |
| Vargas | 2500 |

```
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 |

第二单元：条件限制和排序

使用LIKE做模糊匹配：

可使用% 或者_ 作为通配符：

% 代表 0个或者多个 字符.

_ 代表一个单个字符.

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

| LAST_NAME |
|-----------|
| Kochhar |
| Lorentz |
| Mourgos |

.....

第二单元：条件限制和排序

如果要搜索通配符本身该怎么办呢？：

```
SQL> select * from t_char;
```

A

a_b

acb

a%b

a'b

a/b

a\b

%

—

a

要求找出含有%的记录

这需要使用ESCAPE 标识转义字符

```
select * from t_char where a like '%\%%' escape '\';
```

测试：下面这样写是否也可以？

```
select * from t_char where a like '%K%%' escape 'K';
```

第二单元：条件限制和排序

使用NULL条件：

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

| LAST_NAME | MANAGER_ID |
|-----------|------------|
| King | |

使用多个条件组合的逻辑操作符：

| 逻辑操作符 | 意义 |
|-------|-------------------|
| AND | 所有条件都满足，返回TRUE |
| OR | 只要有一个条件满足，返回TRUE |
| NOT | 如果条件是FALSE,返回TRUE |

```
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 |

.....

第二单元：条件限制和排序

```
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 |

```
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 |

.....

第二单元：条件限制和排序

使用ORDER BY 子句进行排序：

ASC：升序

DESC：倒序

```
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 |

.....

```
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 |

第二单元：条件限制和排序

按照字段别名排序：

```
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 |

.....

按照 多个字段 排序：

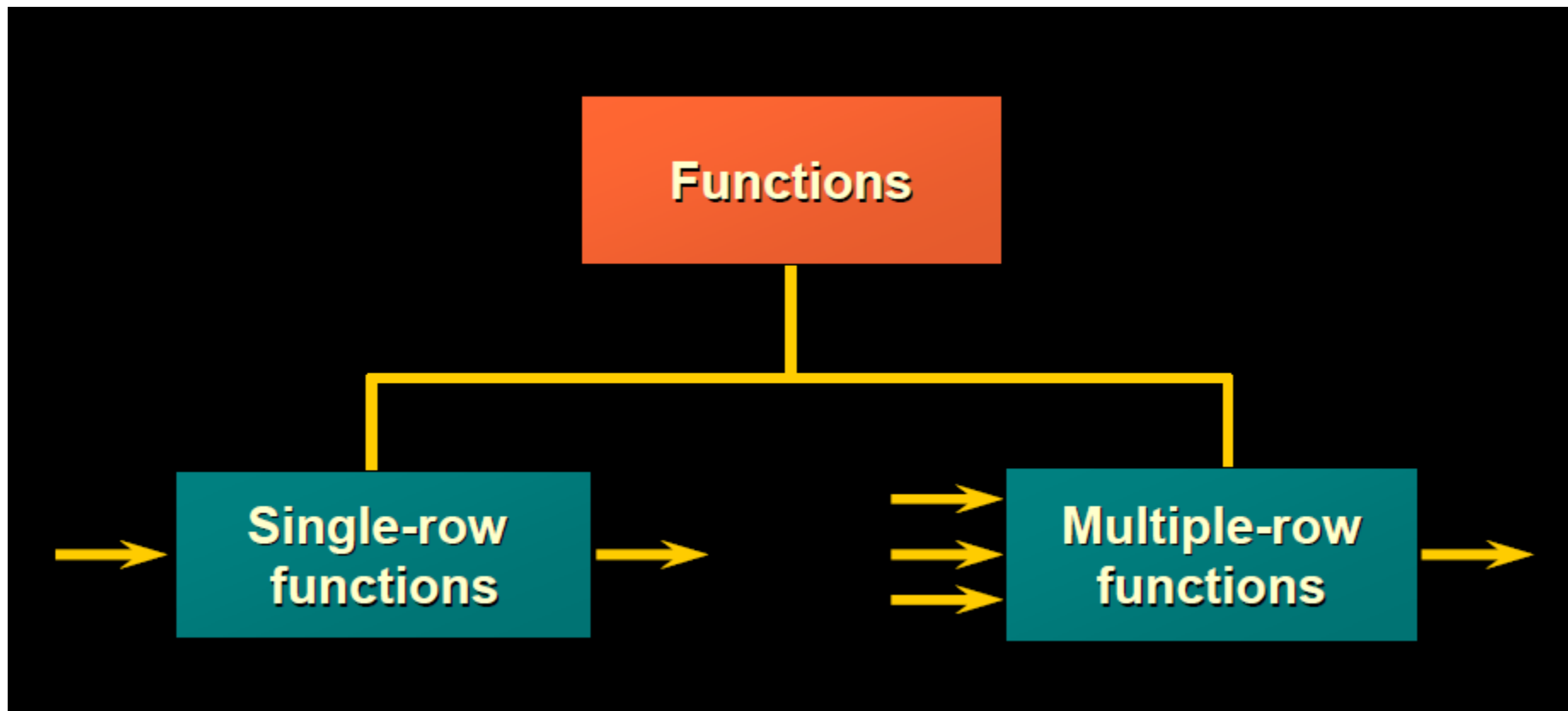
```
SELECT last_name, department_id, salary
FROM employees
ORDER BY department_id, salary DESC;
```

| 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 |

.....

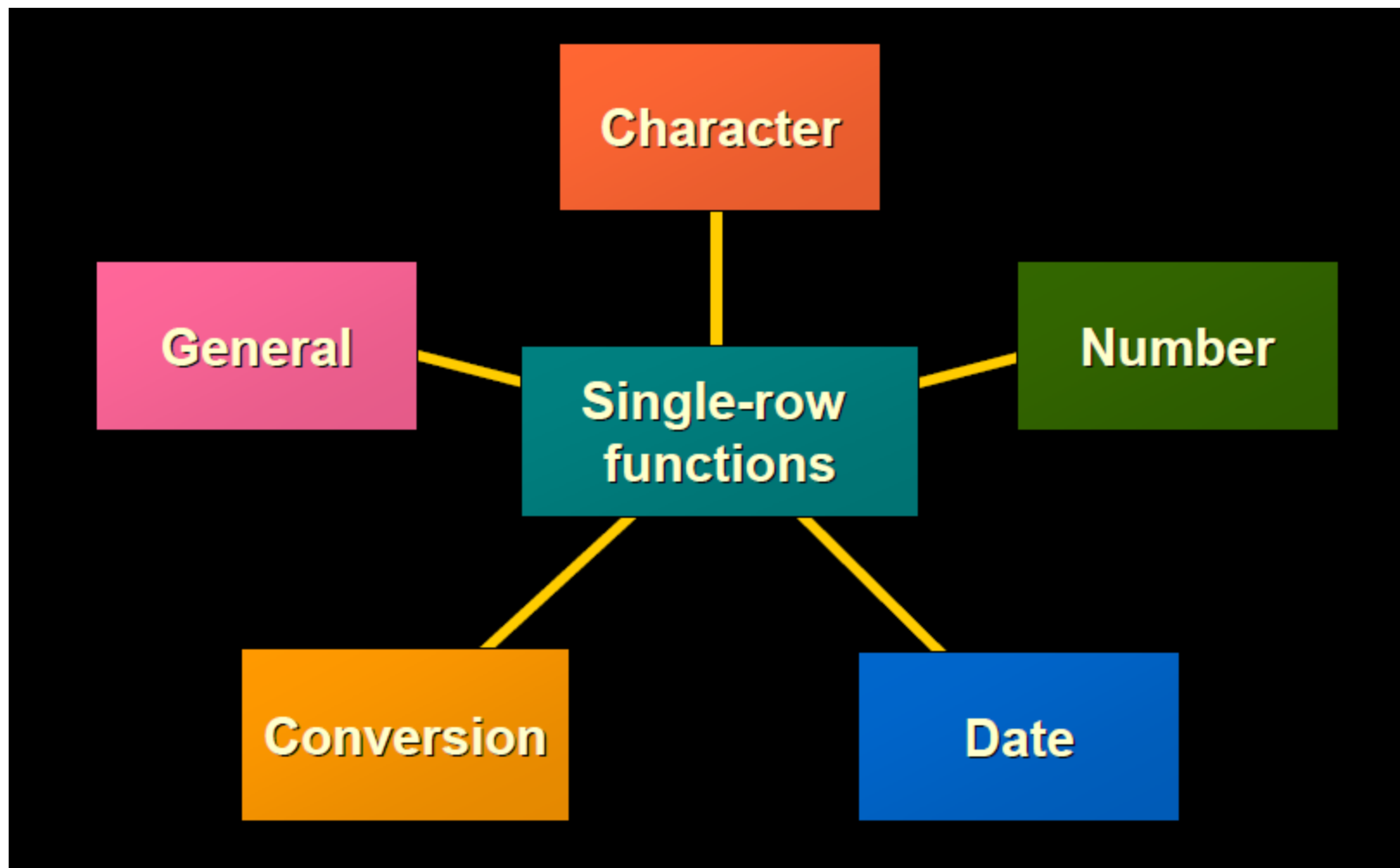
第三单元：单行函数

SQL函数类型：单行函数和多行函数

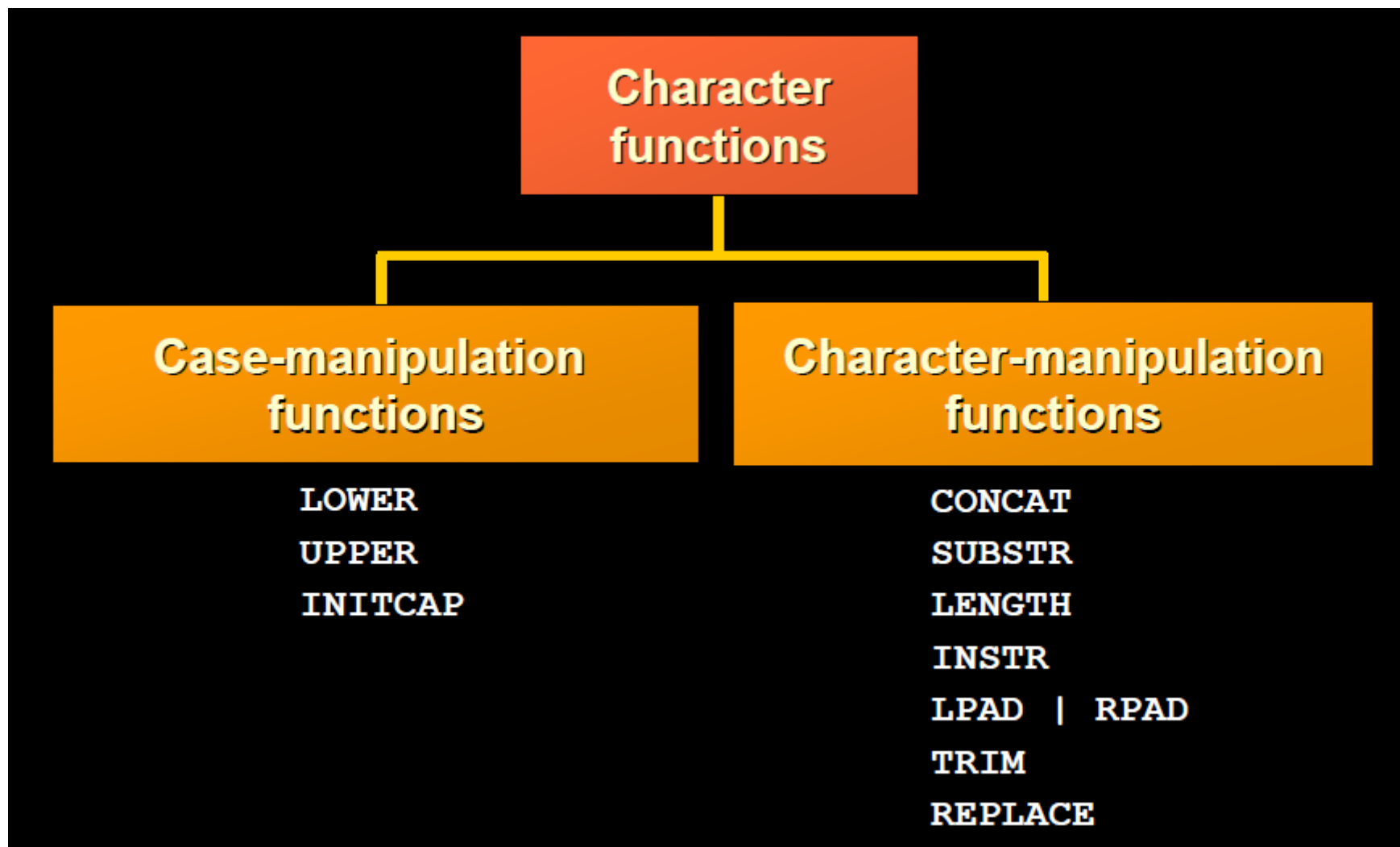


第三单元：单行函数

单行SQL函数类型：



字符函数：



第三单元：单行函数

大小写转换函数：

| 函数 | 结果 |
|-----------------------|------------|
| LOWER('SQL Course') | sql course |
| UPPER('SQL Course') | SQL COURSE |
| INITCAP('SQL course') | Sql Course |

Oracle数据库中的数据是大小写敏感的：

```
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 |

.....

第三单元：单行函数

字符串操作函数：

| 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 |
| TRIM(' ' HelloWorld) | HelloWorld |
| TRIM('Hello World') | Hello World |

SQL语句中使用举例：

```
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 |

第三单元：单行函数

数字操作函数：

| 函数 | 结果 |
|-----------------|-------|
| ROUND(45.926,2) | 45.93 |
| TRUNC(45.926,2) | 45.92 |
| MOD(1600,300) | 100 |

SQL语句中使用举例：

```
SELECT ROUND(45.923,2), ROUND(45.923,0), ROUND(45.923,-1) FROM DUAL;
```

| ROUND(45.923,2) | ROUND(45.923,0) | ROUND(45.923,-1) |
|-----------------|-----------------|------------------|
| 45.92 | 46 | 50 |

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

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

```
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 |
| | | |

第三单元：单行函数

日期操作函数：

| 函数 | 结果 |
|---|--|
| 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') | 8-Sep-95 |
| NEXT_DAY ('01-SEP-95',1) | 3-Sep-95 |
| NEXT_DAY ('1995-09-01',1) | ORA-01861:literal does not match format string |
| NEXT_DAY (to_date('1995-09-01','YYYY-MM-DD'),1) | 3-Sep-95 |
| LAST_DAY('01-FEB-95') | 28-Feb-95 |
| ROUND('25-JUL-95','MONTH') | 1-Aug-95 |
| ROUND('25-JUL-95','YEAR') | 1-Jan-96 |
| TRUNC('25-JUL-95','MONTH') | 1-Jul-95 |
| TRUNC('25-JUL-95','YEAR') | 1-Jan-95 |

Report Window - NLS Database Parameters.rep

| NLS Database Parameters | |
|--------------------------|-----------|
| Parameter | Value |
| NLS_CALENDAR | GREGORIAN |
| NLS_CHARACTERSET | AL32UTF8 |
| NLS_COMP | BINARY |
| NLS_CSMIG_SCHEMA_VERSION | 5 |
| NLS_CURRENCY | \$ |
| NLS_DATE_FORMAT | DD-MON-RR |
| NLS_DATE_LANGUAGE | AMERICAN |
| NLS_DUAL_CURRENCY | \$ |
| NLS_ISO_CURRENCY | AMERICA |

Preferences

Connection: Default jackshang

Oracle

- Connection
- Options
- Compiler
- Debugger
- Output
- Trace
- Profiler
- Logon History
- Hints

User Interface

- Options
- Toolbar
- Object Browser
- Editor
- Fonts
- PL/SQL Beautifier
- Code Assistant
- Key Configuration
- Appearance
- NLS Options

Window Types

- Program Window
- SQL Window
- Test Window
- Plan Window

Tools

Date

yyyy/M/d

☐ User defined

☒ Windows format

☐ Oracle format

Time

h:mm:ss

☐ User defined

☒ Windows format

Numbers

Decimal Symbol: | Digit Group Symbol: |

☐ User defined

☒ Windows format

☐ Oracle format

第三单元：单行函数

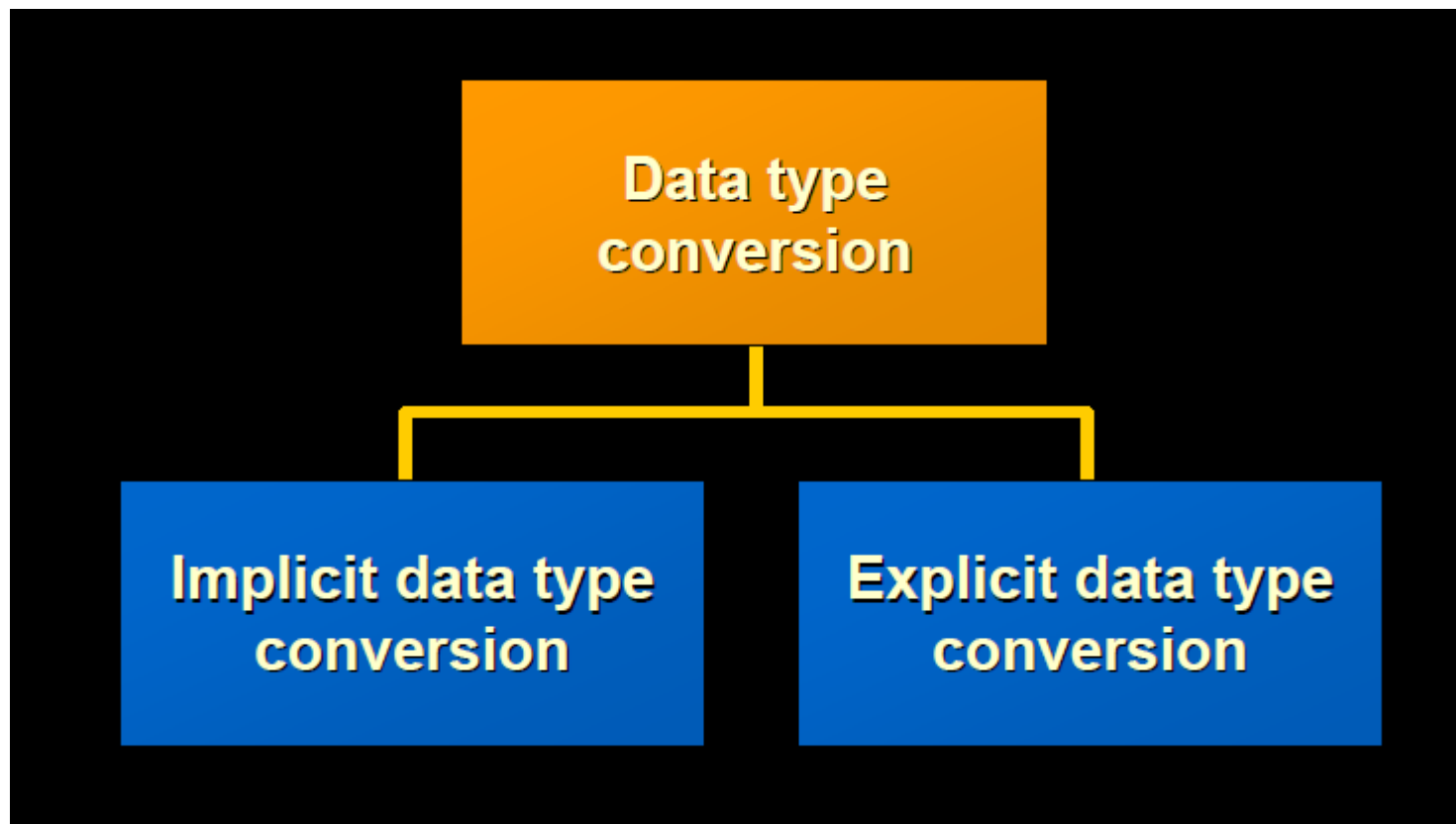
日期的运算操作：

```
SELECT last_name, (SYSDATE-hire_date)/7 AS WEEKS, sysdate+1 as  
tomorrow , hire_date + 8/24  
FROM employees  
WHERE department_id = 90;
```

| | LAST_NAME | WEEKS | TOMORROW | HIRE_DATE+8/24 |
|---|-----------|------------------|--------------------|-------------------|
| 1 | King | 1234.52050760582 | 2011/2/13 15:26:43 | 1987/6/17 8:00:00 |
| 2 | Kochhar | 1116.37765046296 | 2011/2/13 15:26:43 | 1989/9/21 8:00:00 |
| 3 | De Haan | 943.52050760582 | 2011/2/13 15:26:43 | 1993/1/13 8:00:00 |

.....

不同类型的数据转换函数：



第三单元：单行函数

Oracle 数据类型的 隐式转换规则：

对于赋值操作可以：

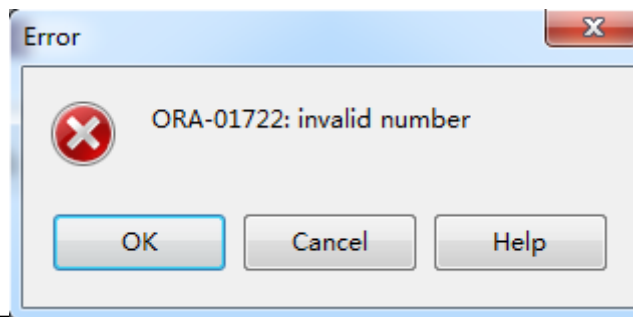
| 从 | 到 |
|------------------|----------|
| VARCHAR2 or CHAR | NUMBER |
| VARCHAR2 or CHAR | DATE |
| NUMBER | VARCHAR2 |
| DATE | VARCHAR2 |

对于表达式比较操作仅可以：

| 从 | 到 |
|------------------|--------|
| VARCHAR2 or CHAR | NUMBER |
| VARCHAR2 or CHAR | DATE |

```
select * from test1;
```

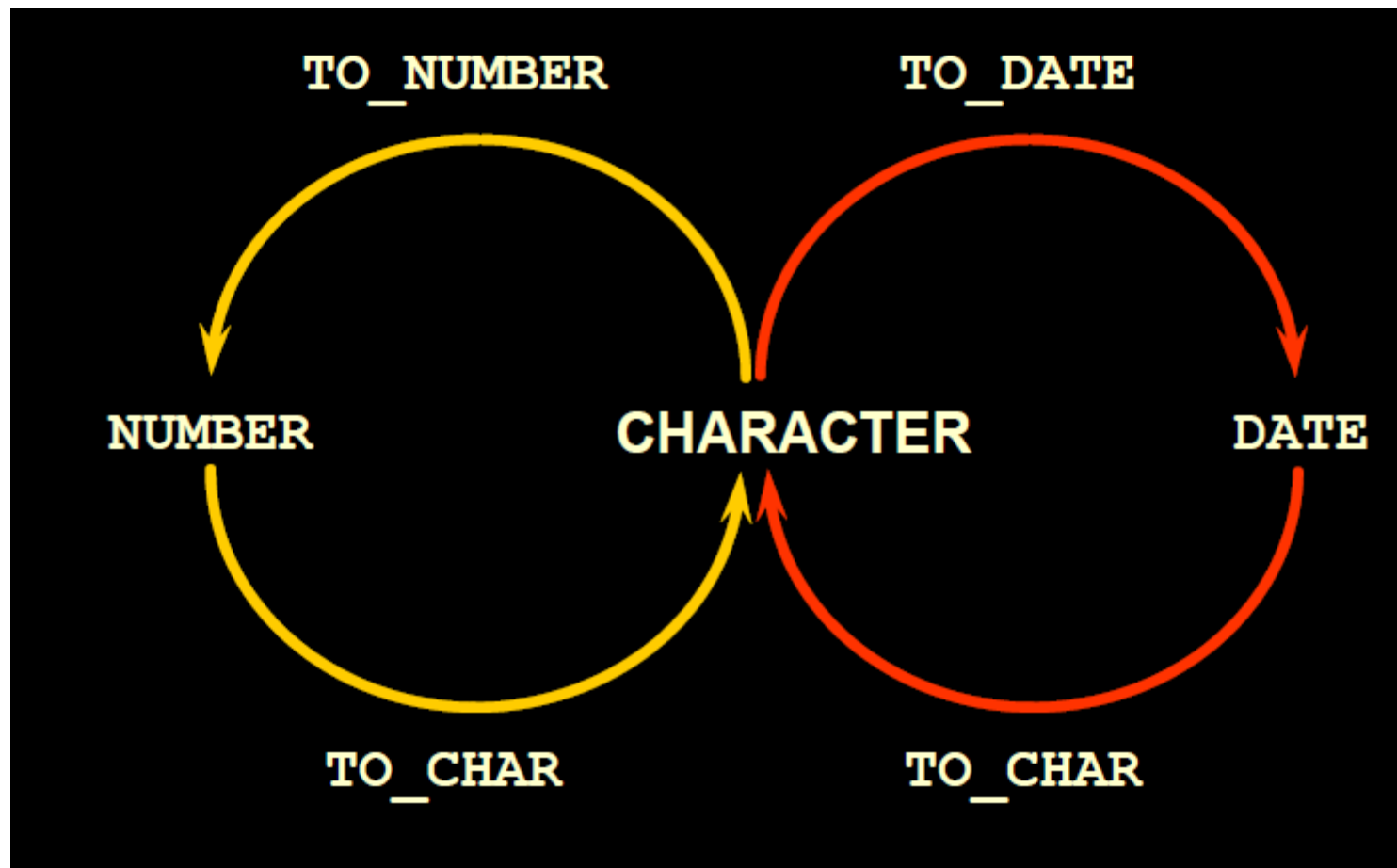
| | COLUMN1 |
|---|---------|
| 1 | 4%56a |
| 2 | 1234 |
| 3 | aaaa |



```
select * from test1 where column1 = 1234;
```

第三单元：单行函数

Oracle 数据类型的 显式转换函数：



第三单元：单行函数

TO_CHAR() 函数：日期到字符串的转换

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

| 日期格式化元素 | 意义 |
|---------------|-----------------|
| YYYY | 4位数字表示的年份 |
| YEAR | 英文描述的年份 |
| MM | 2位数字表示的月份 |
| MONTH | 英文描述的月份 |
| MON | 三个字母的英文描述月份简称 |
| DD | 2位数字表示的日期 |
| DAY | 英文描述的星期几 |
| DY | 三个字母的英文描述的星期几简称 |
| HH24:MI:SS AM | 时分秒的格式化 |
| DDspth | 英文描述的月中第几天 |
| fm | 格式化关键字，可选 |

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

| | LAST_NAME | HIREDATE |
|-------|-----------|----------------------|
| 1 | King | 17 of June 1987 |
| 2 | Kochhar | 21 of September 1989 |
| 3 | De Haan | 13 of January 1993 |
| 4 | Hunold | 3 of January 1990 |
| | | |

第三单元：单行函数

TO_CHAR() 函数：数字到字符串的转换

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

| 数字格式化元素 | 意义 |
|---------|------------|
| 9 | 表示一个数字 |
| 0 | 强制显示0 |
| \$ | 放一个美元占位符 |
| L | 使用浮点本地币种符号 |
| . | 显示一个小数点占位符 |
| , | 显示一个千分位占位符 |

```
alter session set NLS_CURRENCY = '¥';
```

```
SELECT TO_CHAR(salary, 'L99,999.00') SALARY FROM employees  
WHERE last_name = 'Ernst';
```

| | |
|---|---------------|
| | SALARY |
| 1 | ¥6,000.00 |

```
alter session set NLS_CURRENCY = '$';
```

| | |
|---|---------------|
| | SALARY |
| 1 | \$6,000.00 |

再次执行上面的Select 语句会如何？

第三单元：单行函数

TO_NUMBER() 函数：字符串到数字的转换

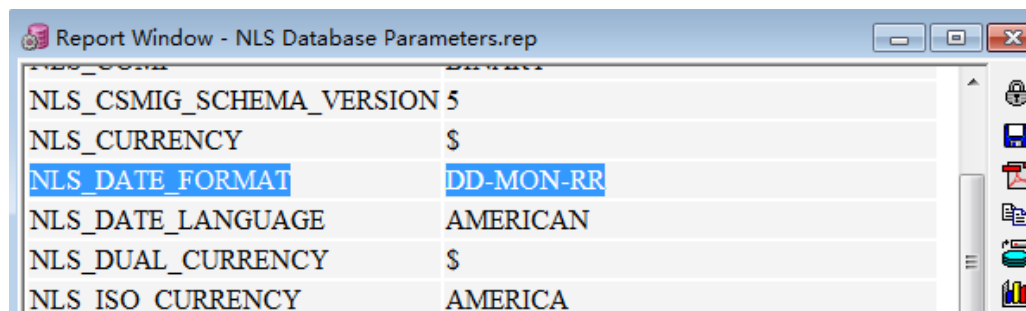
```
TO_NUMBER(char[, 'format_model']) ;
```

| TO_NUMBER应用 | 正确与否 |
|--|------|
| select TO_NUMBER('4456') from dual; | √ |
| select TO_NUMBER('\$4,456') from dual; | X |
| select TO_NUMBER('\$4,456','\$9,999') from dual; | √ |
| select TO_NUMBER('\$4,456,455.000','\$9,999.999') from dual; | X |
| select TO_NUMBER('\$4,456,455.000','\$9,999,999,999,999.999') from dual; | √ |

TO_DATE() 函数：字符串到日期的转换

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

| TO_DATE应用 | 正确与否 |
|--|------|
| select to_date('22-FEB-11') from dual; | √ |
| select to_date('2011-2-22') from dual; | X |
| select to_date('2011-2-22','YYYY-MM-DD') from dual; | √ |
| select to_date('2-22-2011','MM-DD-YYYY') from dual; | √ |
| select to_date('2011-FEB-22','YYYY-MON-DD') from dual; | √ |



第三单元：单行函数

TO_DATE() 函数：日期转换时使用RR格式的注意事项

| Current Year | | Specified Date | RR Format | YY Format |
|--------------|--|----------------|-----------|-----------|
| 1995 | | 27-OCT-95 | 1995 | 1995 |
| 1995 | | 27-OCT-17 | 2017 | 1917 |
| 2001 | | 27-OCT-17 | 2017 | 2017 |
| 2001 | | 27-OCT-95 | 1995 | 2095 |

| | | If the specified two-digit year is: | |
|--|-------|---|--|
| | | 0–49 | 50–99 |
| If two digits of the current year are: | 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 |

第三单元：单行函数

TO_NUMBER() 函数：字符串到数字的转换

比如要从员工信息表中找出入职日期在1990年1月1日以后的的员工，正好适合使用RR格式（因为现在正处于上半个世纪）；

问题：假设现在是2065年，下属SQL还正确吗？应该怎么写？

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

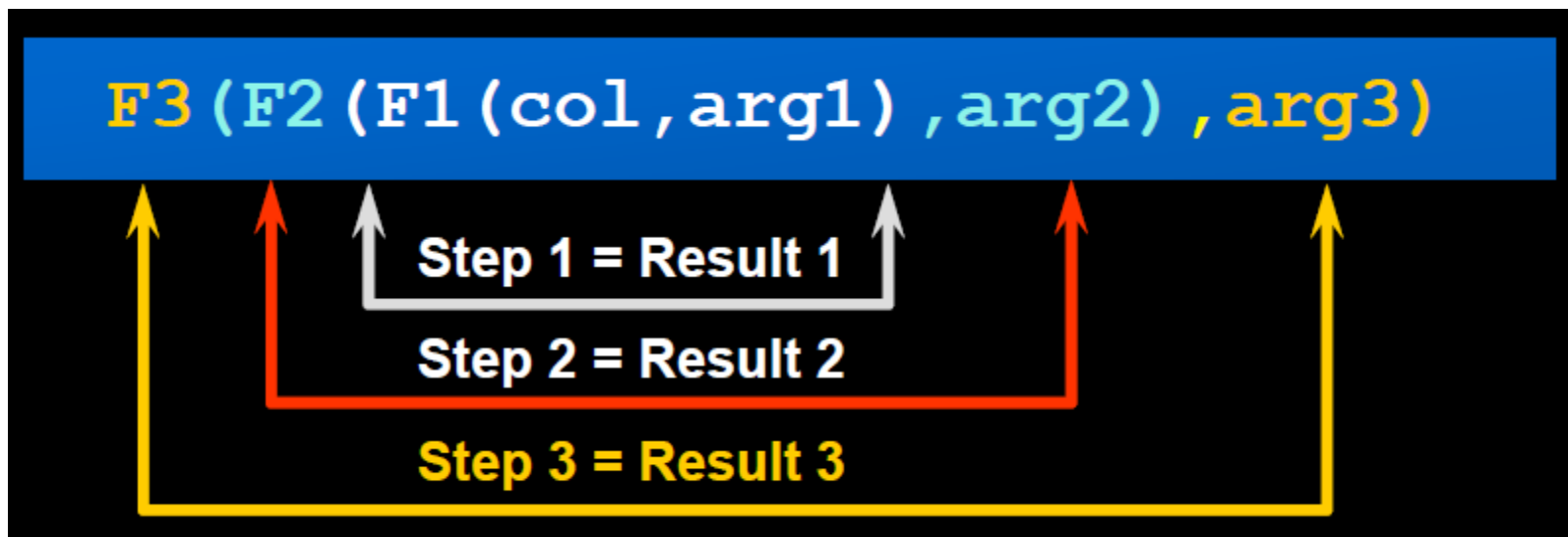
| LAST_NAME | TO_CHAR(HIR |
|-----------|-------------|
| King | 17-Jun-1987 |
| Kochhar | 21-Sep-1989 |
| Whalen | 17-Sep-1987 |

.....

第三单元：单行函数

函数嵌套

单行函数可以被无限层的嵌套，计算时先计算里层，再计算外层



```
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 |

第三单元：单行函数

其他常用单行函数

| 函数 | 用途 |
|-------------------------------------|--|
| NVL (expr1, expr2) | 如果expr1为空，这返回expr2 |
| NVL2 (expr1, expr2, expr3) | 如果expr1为空，这返回expr3（第2个结果）否则返回expr2 |
| NULLIF (expr1, expr2) | 如果expr1和expr2相等，则返回空 |
| COALESCE (expr1, expr2, ..., exprn) | 如果expr1不为空，则返回expr1,结束；否则计算expr2,直到找到一个不为NULL的值 或者如果全部为NULL，也只能返回NULL了 |

举例：

```
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 |

.....

第三单元：单行函数

举例：

```
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 |

```
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 | |

第三单元：单行函数

```
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 | |

```
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 |

第三单元：单行函数

条件表达式：

实现方法：CASE 语句 或者DECODE函数，两者均可实现 IF-THEN-ELSE 的逻辑，相比较而言，DECODE 更加简洁。

CASE 语句：

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

DECODE函数：

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

第三单元：单行函数

举例：

```
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 |

第三单元：单行函数

```
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 |

第四单元：多表关联查询

为了避免冗余信息，表结构设计遵循第三范式，所以在大多数情况下，我们需要从多张表中获取数据。下面的例子是从两张表中分别获取部分信息，形成一个查询结果呈现给用户的

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 |

第四单元：多表关联查询

在执行多表查询时，若未指定链接条件，则结果返回是个笛卡尔乘积：
比如：

```
select employee_id, department_id, location_id from employees,  
departments
```

EMPLOYEES (20 rows)

| 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 rows)

| 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.

**Cartesian
product:
20x8=160 rows**

| 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.

第四单元：多表关联查询

显然大多数情况下，笛卡尔乘积不是我们想要的结果，为了避免笛卡尔乘积，我们一般要在Where子句中提供链接条件，对于链接，通常又包括多种类型：

不同的数据库厂商对链接类型有不同的定义，但国际上有个凌驾于各厂商的工业标准定义（SQL 1999），我们先来看Oracle定义的链接类型：

- 1、等于链接
- 2、不等链接
- 3、外连接（可细分为左外连接、右外连接）
- 4、自链接

第四单元：多表关联查询

“等于链接” 语法：

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

举例：

```
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 |

.....

第四单元：多表关联查询

“不等链接” 语法： 使用不等链接符，包括>，<，!=，between

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

举例：

```
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 |

.....

第四单元：多表关联查询

“外链接” 语法： 包括左外连接，右外连接

```
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 (+);
```

第四单元：多表关联查询

举例：

```
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 |

第四单元：多表关联查询

“自链接”：其实是一种概念，某个table和自己本身链接，比如：table1给另一个“自己”起别名为table2

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

举例：

```
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 |

.....

第四单元：多表关联查询

工业标准定义 (SQL 1999) 的连接类型, Oracle 从9i版本开始提供对SQL 1999的兼容支持：

- 1、交叉连接
- 2、自然链接
- 3、Using 子句
- 4、内连接
- 5、外连接 (全外连接、左外连接、右外连接)

SQL 1999的语法：

```
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)];
```

第四单元：多表关联查询

交叉连接：相当于没有连接条件的多表关联查询，结果是个笛卡尔乘积，实际工作中很少应用到。

举例：

```
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.

第四单元：多表关联查询

自然链接：相当于Oracle的“等于连接”，只不过是让系统自己去找两张表中字段名相同的字段作为“等于连接”条件；（注意如果两个表中有相同的列名，但字段类型不一样，这会引发一个错误）

举例：

locations：

| | LOCATION_ID | STREET_ADDRESS | POSTAL_CODE | CITY | STATE_PROVINCE | COUNTRY_ID |
|---|-------------|-------------------------|-------------|-----------|------------------|------------|
| 1 | 1000 | 1297 Via Cola di Rie | 00989 | Roma | | IT |
| 2 | 1100 | 93091 Calle della Testa | 10934 | Venice | | IT |
| 3 | 1200 | 2017 Shinjuku-ku | 1689 | Tokyo | Tokyo Prefecture | JP |
| 4 | 1300 | 9450 Kamiya-cho | 6823 | Hiroshima | | JP |
| 5 | 1400 | 2014 Jabberwocky Rd | 26192 | Southlake | Texas | US |

departments：

| | DEPARTMENT_ID | DEPARTMENT_NAME | MANAGER_ID | LOCATION_ID |
|---|---------------|-----------------|------------|-------------|
| 1 | 10 | Administration | 200 | 1700 |
| 2 | 20 | Marketing | 201 | 1800 |
| 3 | 30 | Purchasing | 114 | 1700 |
| 4 | 40 | Human Resources | 203 | 2400 |
| 5 | 50 | Shipping | 121 | 1500 |

第四单元：多表关联查询

举例：

```
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 |

相当于：

```
SELECT department_id, department_name,  
location_id, city  
FROM departments, locations  
Where departments.location_id = locations.location_id;
```

第四单元：多表关联查询

Using子句：Using子句可开着是 自然连接 的一种补充功能，我们知道自然连接会让系统自动查找两张表中的所有列名相同的字段，并试图建立“等于连接”；但有的时候我们不期望这么做，而只是期望某个特定的字段作为“等于连接”的条件，这种情况下可以使用Using子句来做限制。

举例：

Employees：

| | EMPLOYEE_ID | FIRST_NAME | LAST_NAME | EMAIL | PHONE_NUMBER | HIRE_DATE | JOB_ID | SALARY | COMMISSION_PCT | MANAGER_ID | DEPARTMENT_ID |
|---|-------------|------------|-----------|----------|--------------|-----------|---------|----------|----------------|------------|---------------|
| 1 | 100 | Steven | King | SKING | 515.123.4567 | 1987/6/17 | AD_PRES | 24000.00 | | | 90 |
| 2 | 101 | Neena | Kochhar | NKOCHHAR | 515.123.4568 | 1989/9/21 | AD_VP | 17000.00 | | 100 | 90 |
| 3 | 102 | Lex | De Haan | LDEHAAN | 515.123.4569 | 1993/1/13 | AD_VP | 17000.00 | | 100 | 90 |
| 4 | 103 | Alexander | Hunold | AHUNOLD | 590.423.4567 | 1990/1/3 | IT_PROG | 9000.00 | | 102 | 60 |
| 5 | 104 | Bruce | Ernst | BERNST | 590.423.4568 | 1991/5/21 | IT_PROG | 6000.00 | | 103 | 60 |

departments：

| | DEPARTMENT_ID | DEPARTMENT_NAME | MANAGER_ID | LOCATION_ID |
|---|---------------|-----------------|------------|-------------|
| 1 | 10 | Administration | 200 | 1700 |
| 2 | 20 | Marketing | 201 | 1800 |
| 3 | 30 | Purchasing | 114 | 1700 |
| 4 | 40 | Human Resources | 203 | 2400 |
| 5 | 50 | Shipping | 121 | 1500 |

想建立自然连接，又想控制只使用department_id作为连接条件怎么办？

第四单元：多表关联查询

```
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 |

.....

第四单元：多表关联查询

内连接：相当于Oracle的“等于链接”，关键字：INNER JOIN。

举例：

```
SELECT employee_id, city, department_name
FROM employees e
INNER JOIN departments d ON d.department_id = e.department_id
INNER 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 |

.....

INNER JOIN 可简写为JOIN，即省去INNER

```
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;
```

第四单元：多表关联查询

外连接：可细分为左外连接、右外连接、全外连接。

举例：左外连接

```
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 | | |

第四单元：多表关联查询

举例：右外连接

```
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 |

问题：这个右外连接怎样改写成相同结果的左外连接？

第四单元：多表关联查询

举例：全外连接

```
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 |

第五单元：分组计算函数和GROUP BY子句

分组计算函数：相对于单行函数，也可称之为多行函数，它的输入是多个行构成得一个行集（这个行集可以是一张表的所有行，也可以是按照某个维度进行分组后的某一组行），而输出都是一个值；

比如我们常见的一些分组计算需求：求某个部门的薪资总和，薪资平均值，薪资最大值等等。

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.

The maximum salary in the EMPLOYEES table.

| MAX(SALARY) |
|-------------|
| 24000 |

第五单元：分组计算函数和GROUP BY子句

分组计算函数(常用)：包括

- 1、求和 (SUM)
- 2、求平均值 (AVG)
- 3、计数 (COUNT)
- 4、求标准差 (STDDEV)
- 5、求方差 (VARIANCE)
- 6、求最大值 (MAX)
- 7、求最小值 (MIN)

SQL中使用分组计算函数 的语法

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

第五单元：分组计算函数和GROUP BY子句

举例：

```
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 |

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

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

备注：MIN, MAX 可用于任何数据类型，但AVG , SUM , STDDEV, VARIANCE仅适用于数值型字段。

第五单元：分组计算函数和GROUP BY子句

COUNT 函数说明：

| 函数用法 | 意义 |
|----------------------|------------------------------|
| COUNT(*) | 返回满足选择条件的所有行的行数，包括值为空的行和重复的行 |
| COUNT(expr) | 返回满足选择条件的且表达式不为空行数。 |
| COUNT(DISTINCT expr) | 返回满足选择条件的且表达式不为空，且不重复的行数。 |

```
Select * from test1;
```

| | COLUMN1 |
|---|---------|
| 1 | 4%56a |
| 2 | 1234 |
| 3 | aaaa |
| 4 | |

问题：

Select count(1) 和 Select count(*), select count(column1) 返回的结果一样吗？ (Y)

Select count(distinct Column1) from test1 如果不用distinct关键字可以怎么改写？

要使得结果与上一局相同，下面两句都可以吗？还是只有1句可以？

select count(column1) from (select column1 from test1 group by column1)

select count(*) from (select column1 from test1 group by column1)

第五单元：分组计算函数和GROUP BY子句

当分组计算函数遇到NULL：

Employees表中存在多条记录，其commission_pct 的值为NULL，那么下面的Sql语句等价于A还是B？

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

A: select (select sum(commission_pct) from employees)/(select count(*) from employees) from dual

B: select (select sum(commission_pct) from employees)/(select count(commission_pct) from employees) from dual

请选择。

A或B中有等价于下面这条语句的么？。

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

第五单元：分组计算函数和GROUP BY子句

使用GROUP BY 子句进行分组：

1、可以按照某一个字段分组，也可以按照多个字段的组合进行分组

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

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

```
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 |

第五单元：分组计算函数和GROUP BY子句

使用GROUP BY 子句进行分组：

2、SELECT 查询语句中同时选择分组计算函数表达式和其他独立字段时，其他字段必须出现在Group By子句中，否则不合法。

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

错误信息：

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

正确的写法应该是：

```
SELECT department_id, count(last_name) FROM employees
GROUP BY department_id;
```

第五单元：分组计算函数和GROUP BY子句

使用GROUP BY 子句进行分组：

3、不能在Where 条件中使用分组计算函数表达式，当出现这样的需求的时候，使用Having 子句。

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

错误信息：

```
WHERE    AVG(salary) > 8000
        *
ERROR at line 3:
ORA-00934: group function is not allowed here
```

正确的写法应该是：

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


第五单元：分组计算函数和GROUP BY子句

使用GROUP BY 子句进行分组：

4、分组计算函数也可嵌套使用。

比如下面的例子可获取最高的部门平均薪水：

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

| MAX(AVG(SALARY)) |
|------------------|
| 19333.3333 |

子查询需求场景：

谁的薪水比Abel高？

主查询：



谁的薪水比**Abel**高？

子查询



Abel的薪水是多少？



第六单元：子查询

语法：

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

举例：

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

| LAST_NAME |
|-----------|
| King |
| Kochhar |
| De Haan |
| Hartstein |
| Higgins |

第六单元：子查询

注意点：

单行比较必须对应单行子查询（返回单一结果值的查询）； 比如= ， >

多行比较必须对应多行子查询（返回一个数据集合的查询）； 比如 IN ， > ANY, > ALL 等

```
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
```

正确写法：

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

第七单元：DML语句

多行比较举例：

```
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';
```

| 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 |

.....

第七单元：DML语句

DML: Data Manipulation Language , 数据操纵语言 ; 简单的说就是SQL中的增、删、改 等语句。

INSERT 语句 :

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

方式一：写出表名+列名

注意：在“列”中，对于不允许为NULL的列，必须写出来；对于允许为NULL的列，可以不写出来
在Value中，对于列中未写出来的列，默认赋予NULL值

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

方式二：仅写出表名

注意：在Value中必须对应写出每个列的值，即使是允许NULL的字段，也必须显式的给出 NULL

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

方式三：从另一个表中 Copy 一行

语法：INSERT INTO *table* [*column* (, *column*)] *subquery*;

注意：在这种方式下，不要使用VALUES 关键字

```
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.

第七单元：DML语句

方式四：使用子查询作为插入目标

```
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.

```
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

WITH CHECK OPTION 可以检查要插入的内容，是否符合目标子查询的Where条件

第七单元：DML语句

UPDATE 语句：

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

方式一：更新符合条件的行中某些列为具体的值

```
UPDATE employees SET      department_id = 70
WHERE  employee_id = 113;
```

1 row updated.

方式二：使用子查询的结果作为更新后的值

```
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.

注意：当存在约束的时候，某些更新可能会失败

```
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
```

第七单元：DML语句

DELETE 语句：

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

举例一：删除某些符合条件的记录

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

举例二：删除一张表中的所有记录

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

如果遇到这种需求，也可以使用TRUNCATE 语句，TRUNCATE TABLE copy_emp，但要注意，TRUNCATE 语句无法回滚，因此除非是单独执行，并非常确认，否则慎用。

注意：当存在约束的时候，某些删除操作可能会失败

```
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
```

第七单元：DML语句

MERGE 语句： 比较整合语句， 语法：

```
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);
```

举例：

```
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);
```

第七单元：事务处理

数据一致性的重要意义举例，银行转帐： A 转500元给B，实际上发生了3句DML语句

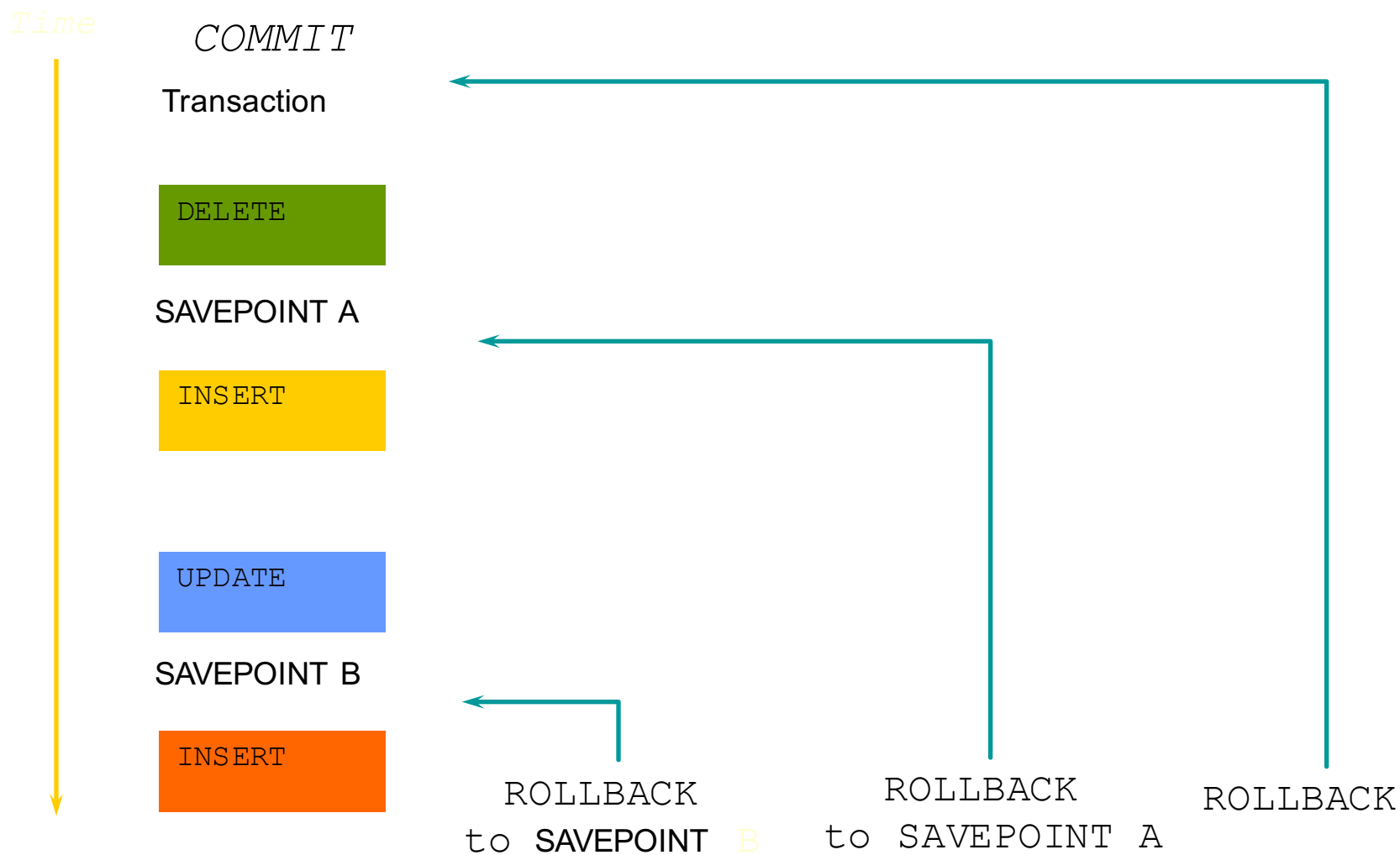
- 1、 10 : 10 : 10.001 : UPDATE A帐户 SET 余额=余额-500
- 2、 10 : 10 : 10.002 : INSERT 转帐交易历史记录
- 3、 10 : 10 : 10.003 : UPDATE B帐户 SET 余额=余额+500

这三个操作必须同时完成，如果有一条部门完成，那么其他也必须回滚，否则数据就不一致。

ORACLE数据库通过事务来控制数据的一致性，当用户进程或者系统崩溃的时候，事务可提供用户更多的灵活性和控制，以确保数据的一致性。

第七单元：事务处理

ORACLE事务来的过程：



第七单元：事务处理

隐式的事务提交或回滚动作：

Commit, rollback 是显式的提交和回滚语句，还有一些隐式的提交和回滚是大家需要知道并引起注意的：

当如下事件发生是，会隐式的执行Commit动作：

- 1、数据定义语句被执行的时候，比如新建一张表：Create Table ...
- 2、数据控制语句被执行的时候，比如赋权 GRANT ... (或者 DENY)
- 3、正常退出 iSQL*Plus 或者PLSQL DEVELOPER, 而没有显式的执行 COMMIT 或者 ROLLBACK 语句。

当如下事件发生时，会隐式执行Rollback 动作：

- 1、非正常退出 iSQL*Plus , PLSQL DEVELOPER, 或者发生系统错误。

课后实验：

1) 使用Plsql Developer作为开发工具，在数据库中创建一张表Testtab1， 往里面插入一条记录，不要commit, 然后正常退出PLSQL，再次登陆看看是否已经执行commit动作。

2) 使用Plsql Developer作为开发工具，在数据库中创建一张表Testtab1， 往里面插入一条记录，不要commit, 然后在任务管理器中将PLSQL进程杀死，再次登陆看看是否已经执行commit动作。

第七单元：事务处理

在Commit 或者 Rollback前后数据的状态：

- 1、在数据已经被更改，但没有Commit前，被更改记录处于被锁定状态，其他用户无法进行更改；
- 2、在数据已经被更改，但没有Commit前，只有当前Session的用户可以看到这种变更，其他Session的用户看不到数据的变化。
- 3、在数据已经被更改，并且被Commit后，被更改记录自动解锁，其他用户可以进行更改；
- 4、在数据已经被更改，并且被Commit后，其他Session的用户再次访问这些数据时，看到的是变化后的数据。

那么同理可知Rollback前后数据的状态及锁的变化。

问题思考：

如果table 1 里面有1亿条数据，完成 delete from table1 需要10分钟，但是在第5分钟的时候，服务器意外关闭，请问再次启动服务器后，table1里面的数据有多少？

- A: 还是1亿条
B: 小于1亿条

读一致性：

用户对数据库的访问无非是两种情况

- 1、读数据 ： Select 语句
- 2、写数据 ： Insert 、 Delete、 Update

如果A用户要读的数据，B用户正在改，那么是否要等B改完再读出来呢？ 如何保证在及其相近的时间内，让多个用户读到一致的数据呢？（假设有用户正在改这一批数据的过程中，尚未提交。。。）

Oracle的“读一致性”概念是指：

- 1、在任何时候，确保提供数据的一致性视图。
- 2、一个用户对数据的更改不会影响另一个用户对数据的更改。
- 3、“读一致性” 确保在同一时刻：
 - 3.1 读数据的人不需要等待写数据的人
 - 3.2 写数据的人不需要等待读数据的人

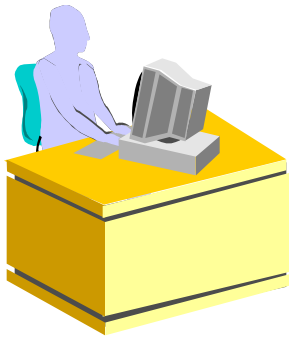
而“读一致性”的通俗理解就是：对于有人正在修改过程中的一批数据，在其提交前，其他用户读到的是一致的内容。

那么, Oracle是如何实现的呢？

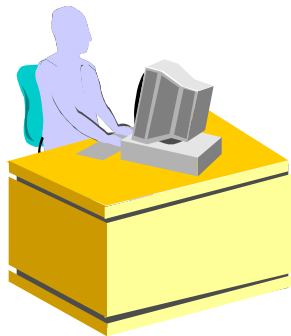
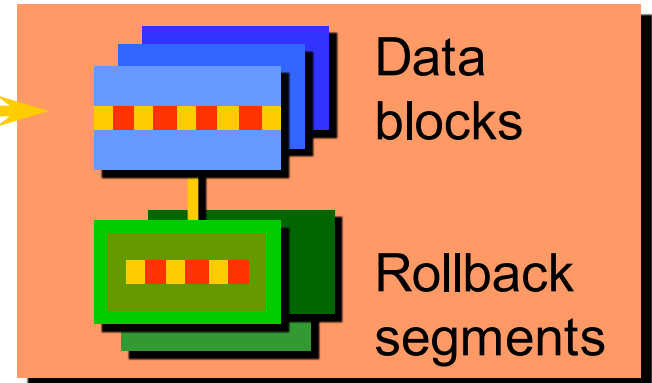
第七单元：事务处理

读一致性实现原理：

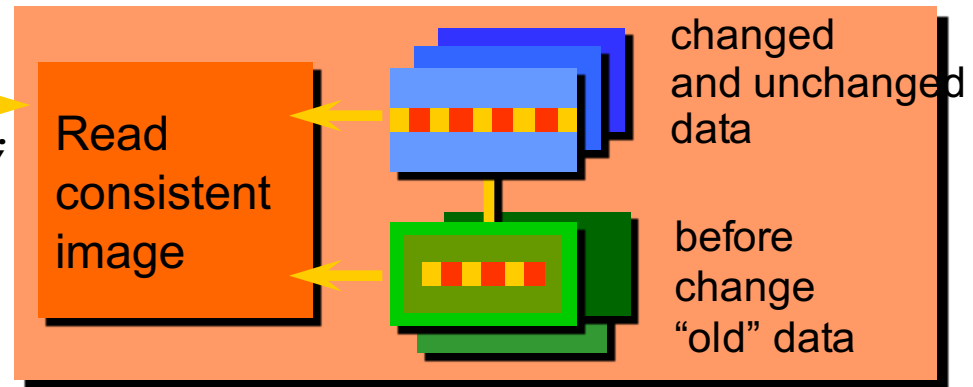
User A



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



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



User B

当有用户修改数据时，Oracle先把那部分原始数据备份到回滚段，在Commit之前，其他Session用户读到的这部分数据是回滚段上的；在提交之后，回滚段被释放。

第八单元：锁

锁的概念：

Oracle中的锁的主要作用就是：防止 并发事务对相同的资源（所谓资源是指 表、行、共享的数据结构、数据字典行等）进行更改的时候，相互破坏。

锁有既有隐式的，也有显式的；但某用户对某一批数据进行更改，而未提交之前，Oracle会隐式的进行加锁；

当然用户也可以显式的加锁，比如：`Select ... from TableA Where ... For UPDATE NoWait`

```
create table testtab3
( Pk1 number ,
  field1 varchar2(200)
);

ALTER TABLE testtab3 ADD CONSTRAINT testtab3_PK PRIMARY KEY(Pk1) ;

insert into testtab3 values (1, 'AAA');
```

先不要提交

查锁

```
select a.*, C.type, C.LMODE
  from v$locked_object a, all_objects b, v$lock c
 where a.OBJECT_ID = b.OBJECT_ID
       and a.SESSION_ID = c.SID
       and b.OBJECT_NAME = 'TESTTAB3'
```

| | XIDUSN | XIDSLOT | XIDSQN | OBJECT_ID | SESSION_ID | ORACLE_USERNAME | OS_USER_NAME | PROCESS | LOCKED_MODE | TYPE | LMODE |
|---|--------|---------|--------|-----------|------------|-----------------|--------------|-----------|-------------|------|-------|
| 1 | 3 | 33 | 230805 | 406969 | 12 | APPS | jackshang | 1544:3752 | 3 TX | | 6 |
| 2 | 3 | 33 | 230805 | 406969 | 12 | APPS | jackshang | 1544:3752 | 3 TM | | 3 |

第八单元：锁

Oracle 的锁是一门学问，涉及到较多知识，希望在今后的工作中做进一步学习。要注意在有主键的表中，由于程序问题导致Insert 相同主键会导致死锁的可能情形：

接上面的例子，另起一个Session，再执行：

```
insert into testtab3 values (1, 'BBB');
```

你会发现，死锁发生了，使用下面的语句查Session之间的阻塞关系

```
SELECT decode(request, 0, 'Holder: ', 'Waiter: ') || sid sess,  
       id1,  
       id2,  
       lmode,  
       request,  
       TYPE  
FROM   gv$lock  
WHERE  (id1, id2, TYPE) IN (SELECT id1, id2, TYPE  
                           FROM   gv$lock  
                           WHERE  request > 0  
                           AND    TYPE != 'HW')  
  
ORDER BY id1, request;
```

其他在实际工作中遇到的“锁相关”的故事：<http://blog.retailsolution.cn/archives/371>

第九单元：数据库对象-表

表的命名要求和表中列的命名要求：

- 1、必须以字母开头
- 2、长度不能超过30个字符
- 3、只能包含 A - Z, a - z, 0 - 9, _, \$, and #
- 4、不能与数据库中的已有对象重名
- 5、不能使用Oracle 数据库的保留字

建表语句的语法：

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

第九单元：数据库对象-表

| 数据类型 | 描述 |
|------------------------|------------------------------------|
| VARCHAR2(size) | 可变长字符串 |
| CHAR(size) | 定长字符串 |
| NUMBER(p,s) | 可变长数值 |
| DATE | 日期时间 |
| LONG | 可变长大字符串，最大可到2G |
| CLOB | 可变长大字符串数据，最大可到4G |
| RAW and LONG RAW | 二进制数据 |
| BLOB | 大二进制数据，最大可到4G |
| BFILE | 存储于外部文件的二进制数据，最大可到4G |
| ROWID | 64进制18位长度的数据，用以标识行的地址 |
| TIMESTAMP | 精确到分秒级的日期类型（9i以后提供的增强数据类型） |
| INTERVAL YEAR TO MONTH | 表示几年几个月的间隔（9i以后提供的增强数据类型-极其少见） |
| INTERVAL DAY TO SECOND | 表示几天几小时几分几秒的间隔（9i以后提供的增强数据类型-极其少见） |

```
CREATE TABLE time_example
(order_date1 TIMESTAMP,
 order_date2 TIMESTAMP WITH TIME ZONE,
 order_date3 TIMESTAMP WITH LOCAL TIME ZONE);
```

```
INSERT INTO time_example VALUES('15-NOV-00 09:34:28', '15-NOV-00 09:34:28 AM -8:00',
                                '15-NOV-00 09:34:28 AM');
```


第九单元：数据库对象-表

从一个子查询快速建表的语法：

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

常用于复制表，比如：

```
CREATE TABLEA as select * from tableb
```

如果只想保留表结构，但不想要数据，可以：

```
CREATE TABLEA as select * from tableb where 1=2
```

第九单元：数据库对象-表

更改表的语法：

添加列：

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

更改列：

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

删除列：

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

第九单元：数据库对象-表

删除表：

```
DROP TABLE tableName;
```

注意：表被删除后，任何依赖于这张表的视图、Package等数据库对象都自动变为无效：

更改表名：

```
RENAME oldtablename to newtableName;
```

一次性清空一张表中的所有内容，但保留表结构：

```
TRUNCATE TABLE tableName;
```

注意TRUNCATE 与DELETE FROM table 的区别：1) 没有Rollback机会 2) HWM标记复位

第十单元：数据库对象-约束

约束的概念：Oracle 数据库使用“约束”来阻止对数据库表中数据的不合法的“增删改”动作。

常用的约束有如下几种：

| | |
|-------------|-----------|
| NOT NULL | (非空约束) |
| UNIQUE | (唯一性约束) |
| PRIMARY KEY | (主键约束) |
| FOREIGN KEY | (外键约束) |
| CHECK | (自定义约束) |

约束的创建方法：

- 1、在创建表的时候同时创建约束
- 2、另外单独创建约束

第十单元：数据库对象-约束

1、在创建表的时候同时创建约束语法：

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

举例：

```
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));
```

第十单元：数据库对象-约束

1、单独创建约束语法：

```
ALTER TABLE tablename ADD CONSTRAINT constraintname  
        constrainttype (column1,...);
```

例子：

```
ALTER TABLE CUX_LES_JE_LINES ADD CONSTRAINT CUX_LES_JE_LINES_PK  
        PRIMARY KEY (JE_LINE_ID);
```


第十单元：数据库对象-约束


常用约束详解：

| | |
|-------------|-----------|
| NOT NULL | (非空约束) |
| UNIQUE | (唯一性约束) |
| PRIMARY KEY | (主键约束) |
| FOREIGN KEY | (外键约束) |
| CHECK | (自定义约束) |

非空约束举例：

```
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,  
    ...
```

 **System
named**

 **User
named**

第十单元：数据库对象-约束

唯一性约束举例：

```
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))
```

 **UNIQUE constraint**

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 |

 **Allowed**
 **Not allowed:**
already exists

第十单元：数据库对象-约束

主键约束举例：

```
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));
```

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 |

...

Not allowed
(Null value)



INSERT INTO

| | | | |
|----|-------------------|-----|------|
| | Public Accounting | | 1400 |
| 50 | Finance | 124 | 1500 |

Not allowed
(50 already exists)



第十单元：数据库对象-约束

外键约束举例：也称为引用数据完整性约束

```
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));
```

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 |

...

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 |

...



INSERT INTO

| | | |
|-----|------|----|
| 200 | Ford | 9 |
| 201 | Ford | 60 |

FOREIGN KEY

Not allowed
(9 does not exist)
Allowed

第十单元：数据库对象-约束

外键约束类型：

- REFERENCES：表示列中的值必须在父表中存在
- ON DELETE CASCADE：当父表记录删除的时候自动删除子表中的相应记录.
- ON DELETE SET NULL：当父表记录删除的时候自动把子表中相应记录的值设为NULL

自定义约束举例：

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

第十单元：数据库对象-约束

删除约束：

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

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

失效/生效 约束：

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

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

第十单元：数据库对象-约束

删除列时，Cascading Constraints 的应用：

```
CREATE TABLE test1 (  
    pk NUMBER PRIMARY KEY,  
    fk NUMBER,  
    col1 NUMBER,  
    col2 NUMBER,  
    CONSTRAINT fk_constraint FOREIGN KEY (fk) REFERENCES test1,  
    CONSTRAINT ck1 CHECK (pk > 0 and col1 > 0),  
    CONSTRAINT ck2 CHECK (col2 > 0));
```

```
ALTER TABLE test1 DROP (pk)
```

ORA-12991: column is referenced in a multicolumn constraint

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

第十单元：数据库对象-约束

查询系统中存在哪些约束：

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

第十一单元：数据库对象-视图

视图的概念：

有的时候我们需要关联多张表获得一个查询结果集，有的时候我们需要写很复杂的条件得到一个想要的结果集，我们不想每次要想这些数据的时候都重新去写很复杂的SQL语句，怎么办？

我们可以把这些结果集创建为视图-View

我们可以把视图分为简单视图和复杂视图两类，主要区别如下：

| 特性 | 简单视图 | 复杂视图 |
|--------------|------|-------|
| 关联的表数量 | 1个 | 1个或多个 |
| 查询中包含函数 | 否 | 是 |
| 查询中包含分组数据 | 否 | 是 |
| 允许对视图进行DML操作 | 是 | 否 |

第十一单元：数据库对象-视图

视图的创建语法：

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

简单视图举例：

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

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;
```


第十一单元：数据库对象-视图

删除视图举例：

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

TOP-N 查询：

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

第十二单元：数据库对象-序列、索引、同意词

序列的概念：

有的时候我们定义某一张表中某一列为主键，当我们往表中插入数据的时候，对于主键字段的赋值要求唯一性，我们希望能有个自增长类型的数据库对象，我们每获取一次，它自动增长，保证下次获取时肯定是不一样的值，这样我们就方便了，Oracle 数据库提供“序列”这种对象来满足我们的要求。

序列的创建：

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

第十二单元：数据库对象-序列、索引、同意词

从序列取值：CURRVAL 取当前值，NEXTVAL取下一个值

序列使用举例：

```
INSERT INTO departments(department_id,  
                        department_name, location_id)  
VALUES      (dept_deptid_seq.NEXTVAL,  
            'Support', 2500);  
  
SELECT      dept_deptid_seq.CURRVAL  
FROM        dual;
```

更改序列定义：

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

删除序列：

```
DROP SEQUENCE dept_deptid_seq;
```

第十二单元：数据库对象-序列、索引、同意词

索引的概念：

但我们的表中数据很多的时候（比如有1亿条数据），我们想找出一条符合特定条件的记录就会比较慢，这个时候，我们希望表中的数据是有序的，这样我们可以使用诸如二分法之类的方法加快查询，而不是做全表扫描，但我们每次要查询的数据可能来自不同的列，我们也无法保证插入表中的数据就是有序的，怎么办呢？Oracle数据库提供“索引”来解决这个问题。

我们可以根据我们查询条件要比较的列来创建“索引”，从索引开始查找。

TABLE

| rowid | Column1 | Column2 | Column3 |
|-------------------|----------|---------|---------|
| AABjWzAAcAAjR4AAA | 1 jack | class1 | |
| AABjWzAAcAAjR4AAB | 3 allen | class5 | |
| AABjWzAAcAAjR4AAC | 2 kate | class4 | |
| AABjWzAAcAAjR4AAD | 7 eric | class2 | |
| AABjWzAAcAAjR4AAE | 5 lilian | class1 | |
| AABjWzAAcAAjR4AAF | 4 tomas | class7 | |
| AABjWzAAcAAjR4AAG | 6 bill | class4 | |
| ... | ... | | |

在Column1上建立的索引| TABLE_INDEX1

| rowid | Column1 |
|-------------------|---------|
| AABjWzAAcAAjR4AAA | 1 |
| AABjWzAAcAAjR4AAC | 2 |
| AABjWzAAcAAjR4AAB | 3 |
| AABjWzAAcAAjR4AAF | 4 |
| AABjWzAAcAAjR4AAE | 5 |
| AABjWzAAcAAjR4AAG | 6 |
| AABjWzAAcAAjR4AAD | 7 |

```
Select * from table
where column1=4;
```

第十二单元：数据库对象-序列、索引、同意词

索引创建举例：

```
CREATE INDEX      emp_last_name_idx  
ON                employees (last_name);
```

Index created.

在什么样的情况下创建索引对加快查询有利呢：

答：查询条件中使用到这个列（或者这个列于其他列的组合），且这个列（或者与其他列的组合）上的数字范围跨度很大，而大多数情况下我们要获取的数据的量占整个表的数据总量 小于4%；

在什么样的情况下不适合创建索引呢：

答：1）被查询的表本身就很很小，即是是全表扫描也非常快；或者基于这张表的查询，大多数情况下需要获取的数据量都超过了总量的4%；或者这张表需要频繁的被更新，建立索引的话会引起索引的频繁更新，从而反而降低数据库的整体效率。

第十二单元：数据库对象-序列、索引、同意词

函数索引：

当查询语句的Where条件中，对于某些列使用了函数表达式时，普通索引对查询没有帮助，如果想利用索引，则必须创建函数索引，比如在下面的例子中，

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

对于上述查询语句，如果建立普通索引，比如Create index dp_idx2 on departments(department_name)

那么上述SQL执行的时候，Oracle是不会走索引的，需要建立函数索引：

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

第十二单元：数据库对象-序列、索引、同意词

课堂试验：插入更多的数据后 看执行计划

```
begin

  FOR i IN 1000 .. 9999 LOOP

    insert into departments values (i,'Test_department_name',200,1700);

  end loop;

end;

EXECUTE DBMS_STATS.GATHER_TABLE_STATS('STUDENT1','DEPARTMENTS');

create index DEPT_NAME_IX on DEPARTMENTS (DEPARTMENT_NAME)

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

第十二单元：数据库对象-序列、索引、同义词

同义词的概念：

当数据库用户A要访问数据库用户B中的一张表Table1的时候，需要加前缀

```
Select * from B.table1
```

但我们要通过DB-LINK访问另一个数据库中的某张表的时候我们需要加@后缀

```
Select * from table1@db-link-name
```

为了在程序中能够简化写法，Oracle 提供同义词，也就是可以在A用户下建立一个同义词指向B用户下的Table1，以后访问的时候可以直接访问这个同义词，而不用加前缀了。

```
CREATE SYNONYM Table1 for B.Table1
```

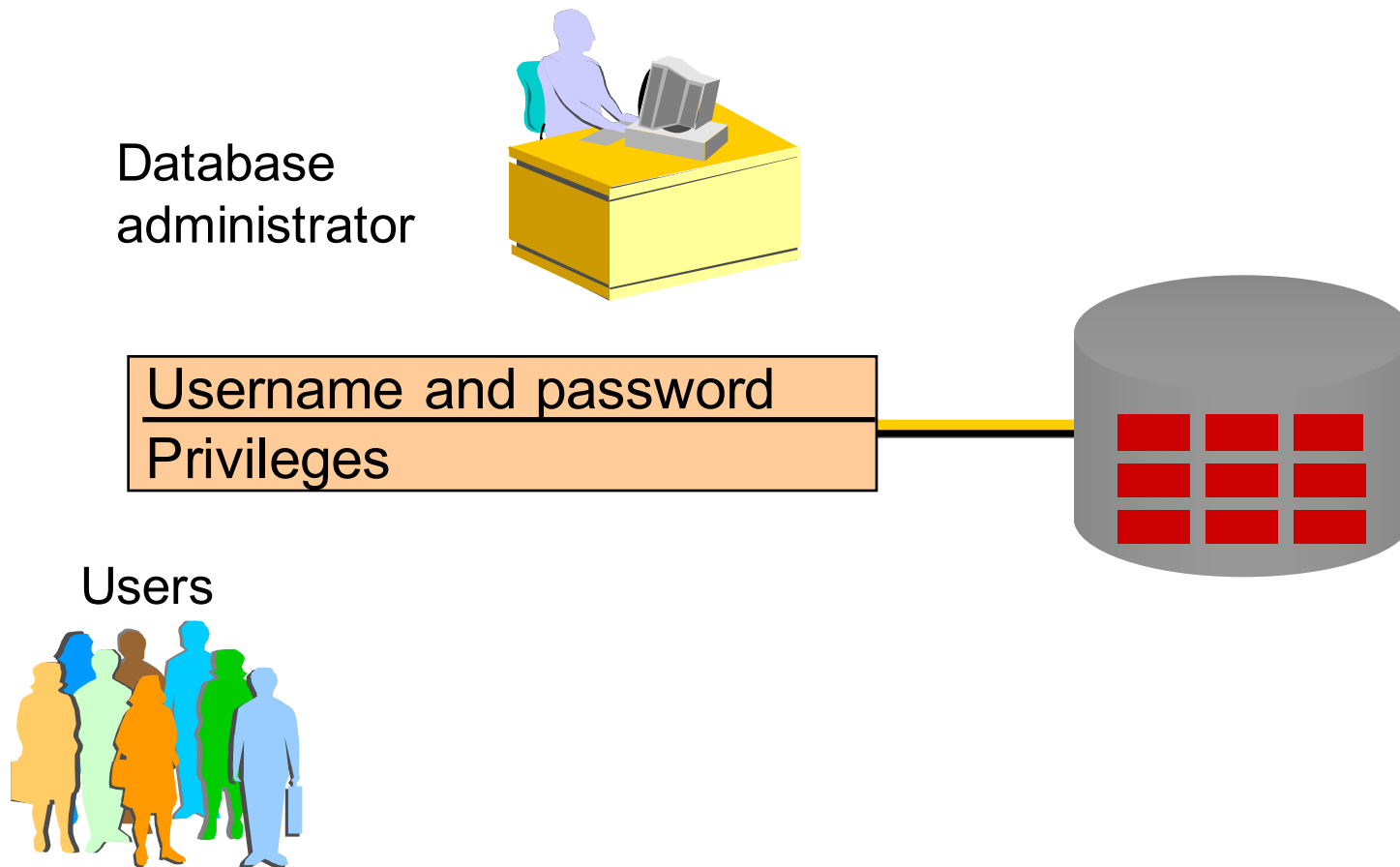
而A用户以后就可以通过同义词访问了

```
Select * from Table1
```


第十三单元：控制用户权限

Oracle的用户权限概念：

每个人登录Oracle数据库都是以某个特定的数据库用户登录的，用户能否创建表？该用户能否访问其他用户下面的表。。等等这些事情都是可以利用Oracle的权限控制机制进行控制的。



第十三单元：控制用户权限

看一个实际例子来理解给用户赋予系统权限：

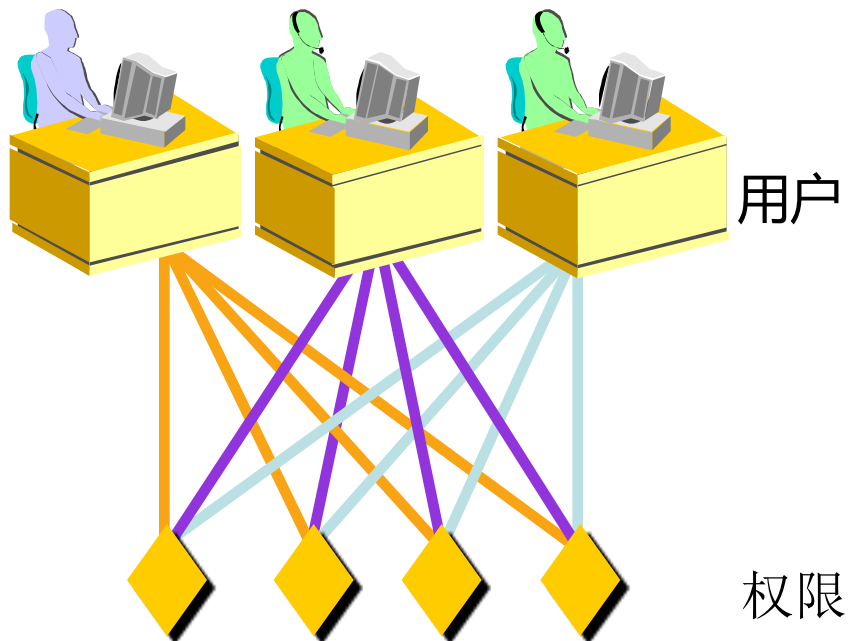
```
REM =====
REM cleanup section
REM =====
REM DROP USER HPOS CASCADE;
spool create_HPOS_schema

REM =====
REM create user
REM =====
CREATE USER HPOS IDENTIFIED BY HPOS;
ALTER USER HPOS DEFAULT TABLESPACE HPOS_DATA QUOTA UNLIMITED ON HPOS_DATA;
ALTER USER HPOS TEMPORARY TABLESPACE temp;
GRANT create session
      , create table
      , create procedure
      , create sequence
      , create trigger
      , create view
      , create synonym
      , alter session
TO HPOS;

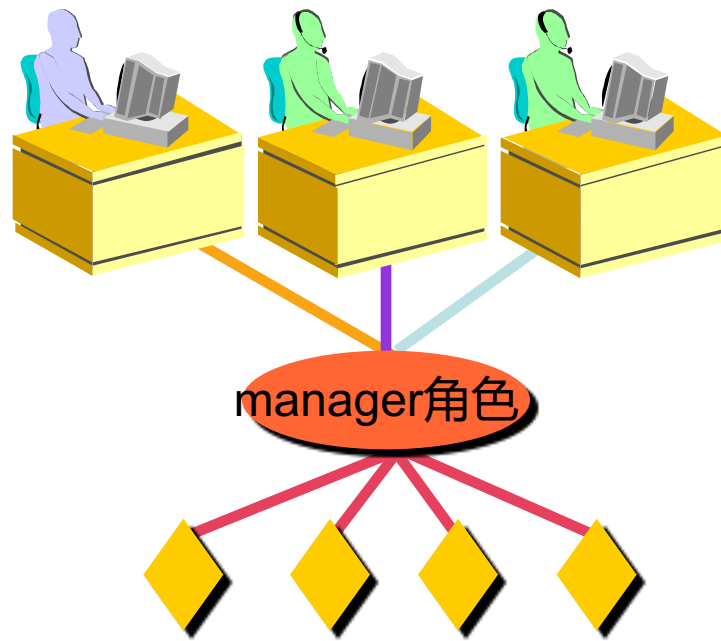
GRANT resource to HPOS;
exit
```

第十三单元：控制用户权限

如果要给多个用户赋予相同的权限，可以通过角色来简化管理：



未通过角色给用户赋权



通过角色给用户赋权

```
CREATE ROLE manager ;  
  
GRANT create table, create view to manager;  
  
GRANT manager to DEHAAN, KOCHHAR;
```

第十三单元：控制用户权限

对象权限：区别于系统权限，细化到某个具体的数据库对象上的权限访问控制，各种数据库对象适合赋予的权限名称列表：

| 对象权限 | Table | View | Sequence | Procedure |
|------------|-------|------|----------|-----------|
| ALTER | √ | | √ | |
| DELETE | √ | √ | | |
| EXECUTE | | | | √ |
| INDEX | √ | | | |
| INSERT | √ | √ | | |
| REFERENCES | √ | √ | | |
| SELECT | √ | √ | √ | |

语法：

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

第十三单元：控制用户权限

普通的对象权限赋权举例：

```
GRANT  update (department_name, location_id)
ON      departments
TO      scott, manager;
```

如果你想让其他用户也有权 把你赋给他的权限进一步赋予给别人，那么需要带 `WITH GRANT OPTION;`

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

如果你想让所有人都有相关权限，那么可以把该权限赋予给Public

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

第十三单元：控制用户权限

通过数据字典查询系统中的赋权情况：

| 数据字典视图 | 描述 |
|---------------------|----------------------|
| ROLE_SYS_PRIVS | 角色对应的系统权限 |
| ROLE_TAB_PRIVS | 角色对应的表权限 |
| USER_ROLE_PRIVS | 用户的角色分配表 |
| USER_TAB_PRIVS_MADE | 用户对象上赋权者与被赋者的历史赋权情况 |
| USER_TAB_PRIVS_RECD | 用户对象上拥有者与被赋者的历史赋权情况 |
| USER_COL_PRIVS_MADE | 用户对象列上赋权者与被赋者的历史赋权情况 |
| USER_COL_PRIVS_RECD | 用户对象列上拥有者与被赋者的历史赋权情况 |
| USER_SYS_PRIVS | 用户的系统权限 |

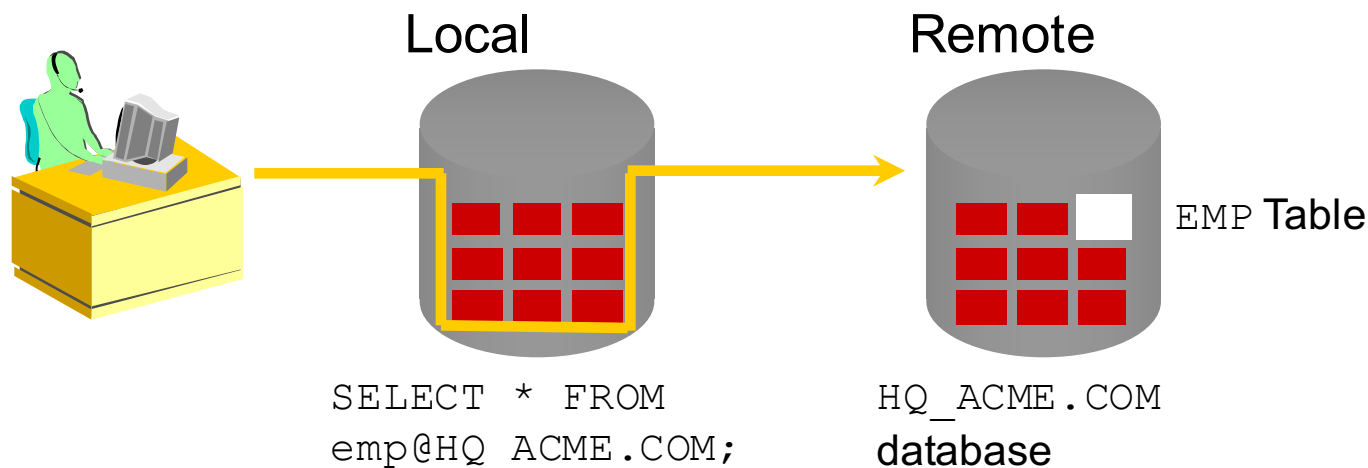
收回权限

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

第十三单元：控制用户权限

数据库连接的概念：

Database Link，如果你需要在当前数据库中访问另一个数据库中表，最简单的方法是在当前数据库中创建一个数据库连接指向另一个数据库，然后通过@数据库连接的后缀就可以访问另一个数据库中的表了。

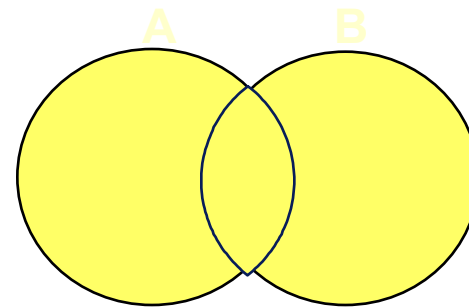
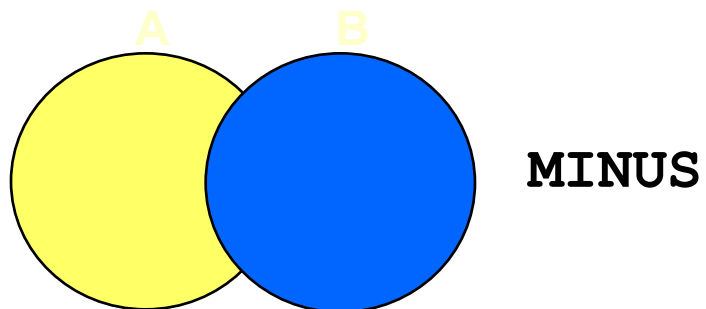
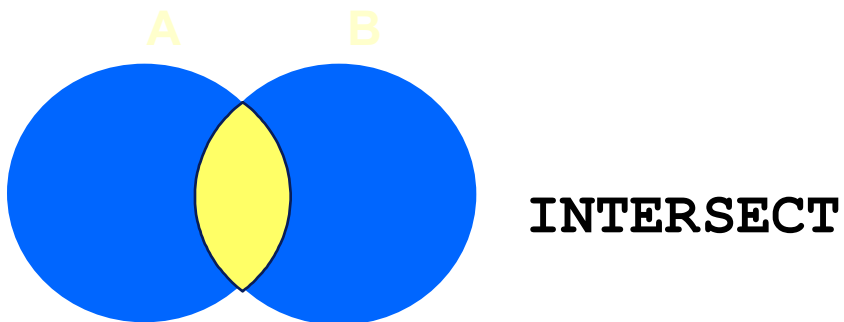
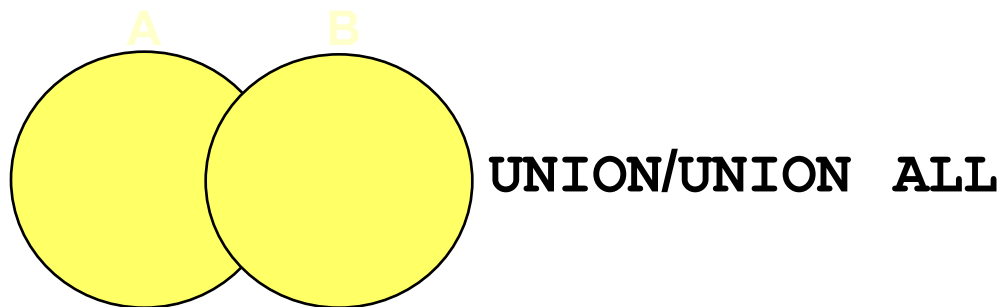


创建 DB-LINK，通过DB-LINK 访问另一数据库中的表

```
CREATE PUBLIC DATABASE LINK hq.acme.com  
USING 'sales';  
Database link created.  
  
SELECT *  
FROM emp@HQ.ACME.COM;
```

第十三单元：控制用户权限

SQL结果集的集合操作：并集、交集、差集



第十三单元：控制用户权限

集合操作举例：

UNION：去除重复记录，
结果中的总记录数可能 < employees的记录数+job_history的记录数

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

UNION ALL 保留重复记录，
结果中的总记录数一定 = employees的记录数+job_history的记录数

```
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;
```

第十三单元：控制用户权限

INTERSECT 取交集

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

MINUS 取差集

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

第十四单元：Group By 子句的增强

在Group By 中使用Rollup 产生常规分组汇总行 以及分组小计：

```
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.

1、常规分组行；2、3、分层小计行；

Rollup 后面跟了n个字段，就将进行n+1次分组，从右到左每次减少一个字段进行分组；然后进行union

第十四单元：Group By 子句的增强

在Group By 中使用Cube 产生Rollup结果集 + 多维度的交叉表数据源：

```
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、常规分组行；2、3、4 分层小计行；其中3是交叉表数据源需要的 job_id 维度层面的小计。
Cube 后面跟了n个字段，就将进行2的N次方的分组运算，然后进行；

第十四单元：Group By 子句的增强

GROUPING函数：Rollup 和 Cube有点抽象，他分别相当于 $n+1$ 和 2^n 的 n 次方常规 Group by 运算；那么在Rollup 和 Cube的结果集中如何很明确的看出哪些行是针对那些列或者列的组合进行分组运算的结果的？ 答案是可以使用Grouping 函数； 没有被Grouping到返回1， 否则返回0

```
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.

第1行， department_id 和 job_id都被用到了，所以都返回0； 第2行， job_id 没有被用到，所以返回1； 第3行， department_id 和 job_id 都没有被用到，所以都返回1

第十四单元：Group By 子句的增强

使用Grouping Set 来代替多次UNION:

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

| DEPARTMENT_ID | JOB_ID | MANAGER_ID | AVG(SALARY) |
|---------------|----------|------------|-------------|
| 10 | AD_ASST | | 4400 |
| 20 | MK_MAN | | 13000 |
| 20 | MK_REP | | 6000 |
| 50 | ST_CLERK | | 2925 |
| | SA_MAN | 100 | 10500 |
| | SA_REP | 149 | 8866.66667 |
| | ST_CLERK | 124 | 2925 |
| | ST_MAN | 100 | 5800 |

26 rows selected.

1

2

第十五单元：子查询进阶

非相关子查询当作一张表来用：

```
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 |

找出所有薪水高于其部门平均薪水的员工：

第十五单元：子查询进阶

相关子查询的概念： 子查询中参考了外部主查询中的表。

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

```
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);
```


第十五单元：子查询进阶

使用Exists操作。

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees outer
WHERE  EXISTS ( SELECT 'X'
                FROM   employees
                WHERE  manager_id =
                      outer.employee_id);
```

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

这两个SQL结果一样，但执行性能是否一样呢？

第十五单元：子查询进阶

使用 Not Exists操作。

```
SELECT department_id, department_name
FROM departments d
WHERE NOT EXISTS (SELECT 'X'
                    FROM   employees
                    WHERE  department_id
                        = d.department_id);
```

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

这两个SQL结果一样吗，同样，请比较一下其执行性能

注意：Not In 里面只要有一个NULL，就不成立了，这是很容易出错的地方；正确的方法请在后面的子查询中加上where department_id is not null;

第十五单元：子查询进阶

在Update 语句中使用相关子查询。

```
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);
```

第十五单元：子查询进阶

在DELETE 语句中使用相关子查询。

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

第十五单元：子查询进阶

使用WITH子句。

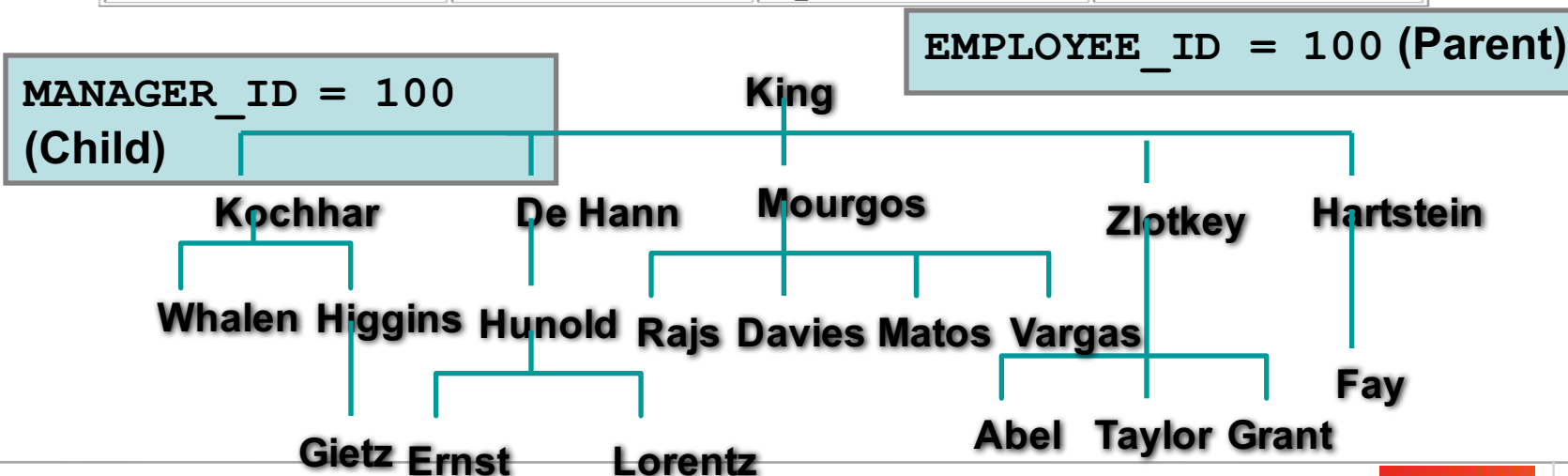
```
WITH
dept_costs AS (
    SELECT  d.department_name, SUM(e.salary) AS dept_total
    FROM    employees e, departments d
    WHERE   e.department_id = d.department_id
    GROUP BY d.department_name),
avg_cost AS (
    SELECT SUM(dept_total)/COUNT(*) AS dept_avg
    FROM    dept_costs)
SELECT *
FROM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

使用WITH好处：1) 如果在后面多次使用则可以简化SQL；2) 适当提高性能

第十六单元：递归查询

EMPLOYEE表：观察EMPLOYEE_ID 和 MANAGER_ID，构成了递归层次关系。

| 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 |



第十六单元：递归查询

递归查询：使用语句SQL语句即可把整个递归树全部查询出来。

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

举例：查询从King开始，从上往下的各级员工。

```
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 ;
```

举例：查询从101开始，从下往上的各级员工。

```
SELECT employee_id, last_name, job_id, manager_id  
FROM   employees  
START   WITH  employee_id = 101  
CONNECT BY PRIOR manager_id = employee_id ;
```

第十六单元：递归查询

查询方向

递归查询的遍历方向。

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

从上往下 \longrightarrow **Column1 = Parent Key**
 Column2 = Child Key

从下往上 \longrightarrow **Column1 = Child Key**
 Column2 = Parent Key

从上往下遍历：

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

从下往上遍历：

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

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


第十六单元：递归查询

使用LEVEL关键字和 LPAD函数 ，在OUTPUT中显示树形层次。

```
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
```

第十七单元：INSERT 增强

一个来源插入多个目标表（无条件）。

```
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;
```

一个来源插入多个目标表（有条件）。

```
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;
```

第十七单元：INSERT 增强

一个来源插入多个目标表（有条件，首次匹配即跳到下一条）。

```
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;
```

列转行（一行变多行，交叉报表的反操作）。

```
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;
```

- 1 有些同学的PLSQL Developer 连接XP虚拟机中的Oracle 10G 数据库时经常断线，需要不断的重新连接，非常麻烦，有何解决方法？

答：导致此问题的原因可能是打开虚拟机的时候选择了 “ I Copyed it” ,从而导致网卡MAC地址不对。

解决方法：

把虚拟机的网络适配器删了从新安装新的：

Step-by-step. 1关了虚拟机服务里的所有oracle服务，并把自动改为手动

Step-by-step. 2我的电脑----管理-----设备管理器-----网络适配器-----右键卸载

Step-by-step. 3在设备管理器里点击工具栏的 ‘扫描硬件改动’ ， ‘更新驱动程序’ 当网络 适配器驱动有显示后，启动oracle服务。

Step-by-step. 4更新驱动程序之后，需要再把虚拟机IP地址改为192. 168. 15. 70, 网关为

192. 168. 15. 2

搞定。

另有同学发现此现象还跟xp虚拟机的电源选项有关，改成 “从不” 会正常。

