

# 0基本构造类语法

## 0.1SQL的基本数据类型

**char(n)** n长度固定的字符串

**varchar(n)** n长度的长度可变字符串

**int** 整数数据类型

**smallint** 小整数，依赖机器的整数集的子集

**numeric (p,d)** 指定精度的定点数，总p位长度，小数点左边p-d位，小数点右边d位

**numeric(p,0)** 也容易表示长度位p的整数

**real , double precision** :单/双精度浮点数

**float(n)** 精度至少为n位数字的浮点数

## 0.2基本建表模式

```
create table department(  
    dept_name varchar(20),  
    building varchar(15),  
    budget numeric(12,2);  
    primary key(dept_name)  
);  
-- 建表的时候指定主键
```

格式

```
create table r(  
    A1 D1,  
    A2 D2,  
  
    A3 D3,
```

.....

<完整约束1> ,

<完整约束2> ,

<完整约束3> ,

.....

);

其中 **primary key** 既是主键

primary key的建表语句(直接设置即可)

primary key (ID);

**foreign key** 既是外键

foreign key 的建表语句

foreign key (dept\_name) references department(dept\_name);

既是自己的某个属性 依赖 另一张表的某个属性值

**not null**的使用

name varchar(20) not null;

既是 直接设立在某个属性后面起说明作用即可

## 0.3实验部分

### 1实验一二

要求:

a.Create Table, Drop Table, Alter Table

b.熟悉主键，外键，check等完整性约束的建立，删除，有效化，无效化

关于主键：

删除主键



```
ALTER TABLE INSTRUCTOR DROP PRIMARY KEY;
```

删除受外键约束的主键



```
ALTER TABLE INSTRUCTOR DROP PRIMARY KEY CASCADE;  
--删除主键，并且删除那些引用了该主键的那些表关于该ID主键的外键
```

使主键失效



```
ALTER TABLE INSTRUCTOR DISABLE PRIMARY KEY;
```

使主键生效




```
alter table tableName enable primary key;
```

增加主键



```
ALTER TABLE INSTRUCTOR ADD PRIMARY KEY (ID);
```


删除主键被引用的表



```
DROP TABLE INSTRUCTOR CASCADE CONSTRAINT;
```

```
-- 删除该表，同时删除以该表为外键的外键，同时删除该表的 索引  
-- 关于索引后续会学到
```

删除约束的时候删除该表的索引



```
ALTER TABLE INSTRUCTOR DROP CONSTRAINT FK_INSTRUCTOR_DEPT_NAME CASCADE  
DROP INDEX;
```

这里引申第二种增加 主键的方式 (CONSTRAINT使用有感)



```
ALTER TABLE INSTRUCTOR ADD CONSTRAINT MINE_PK PRIMARY KEY (ID);
```

注意：第二种方式 较于第一种方式 可以 自定义 约束名称，使用更加灵活，日常键的学习和使用尤其建议第二种方式（也可以软件查看约束名称（**SYS\_** 或 **\_** 开头））

这里特别强调，alter设立约束可以自定义约束名称

如果想要查看建表时系统定义的约束名称：可以通过违反该约束获得，也可以使用navicat中的设计表中查看表的详细结构信息和约束名称等等

可见，关键词：**enable/disable drop add**

关于外键：

外键删除



```
ALTER TABLE INSTRUCTOR DROP CONSTRAINT SYS_C0095022;
```

```
-- 这里是系统命名的 约束名称
```

## 外键添加

```
ALTER TABLE INSTRUCTOR ADD CONSTRAINT FK_DEPT_NAME FOREIGN KEY
(DEPT_NAME) REFERENCES DEPARTMENT(DEPT_NAME);
-- 这里是自定义的约束名称：关于外键的 定于语法见前面
```

## 外键添加和外键的三种行为

```
ALTER TABLE INSTRUCTOR ADD CONSTRAINT FK_DEPT_NAME FOREIGN KEY
(DEPT_NAME) REFERENCES DEPARTMENT(DEPT_NAME) ON DELETE SET NULL;
-- 在含有外键的那个表中当删除相关元素时 其对应引用的表的相关值设置为NULL
```

```
ALTER TABLE INSTRUCTOR ADD CONSTRAINT FK_DEPT_NAME FOREIGN KEY
(DEPT_NAME) REFERENCES DEPARTMENT(DEPT_NAME);
-- 不加on delete语句，就是默认情况，默认下不允许删除
```

```
ALTER TABLE INSTRUCTOR ADD CONSTRAINT FK_DEPT_NAME FOREIGN KEY
(DEPT_NAME) REFERENCES DEPARTMENT(DEPT_NAME) ON DELETE CASCADE;
-- 联级删除，删除引用了该外键的相关项
```

## 外键无效化

```
ALTER TABLE INSTRUCTOR DISABLE CONSTRAINT FK_DEPT_NAME;
```

## 外键有效化

```
ALTER TABLE INSTRUCTOR ENABLE CONSTRAINT FK_DEPT_NAME;
```

## 关于UNIQUE

下面关于unique 唯一性约束



```
ALTER TABLE TAKES ADD CONSTRAINT UNIQUE_TAKES UNIQUE (ID, COURSE_ID,  
SEMESTER, YEAR);  
-- 添加唯一键约束
```

唯一约束无效化



```
ALTER TABLE TAKES DISABLE CONSTRAINT UNIQUE_TAKES;
```

唯一约束删除



```
ALTER TABLE TAKES DROP CONSTRAINT UNIQUE_TAKES;
```

## 关于CHECK

创建check



```
ALTER TABLE instructor  
ADD CONSTRAINT mine_check_salary  
CHECK (salary <= 500000);  
-- 外部创建check约束
```



```
CHECK ( salary > 29000 ) ENABLE,  
-- 内部创建CHECK ENABLE可省略可不省略
```



```
CONSTRAINT MINE_CHECK CHECK (salary > 500000) ENABLE;  
-- 内部命名式创建check
```

无效化CHECK



```
ALTER TABLE INSTRUCTOR DISABLE CONSTRAINT mine_check_salary;
```

删去CHECK



```
ALTER TABLE INSTRUCTOR DROP CONSTRAINT mine_check_salary;
```

下列补充对一个表结构字段进行修改



1、添加字段

```
ALTER TABLE TABLE_NAME ADD COLUMN1 TYPE ADD COLUMN2 TYPE ADD COLUMN3  
TYPE ... ;
```

2、删除字段

```
ALTER TABLE TABLE_NAME DROP COLUMN COLUMN_NAME;
```

```
ALTER TABLE TABLE_NAME DROP (COLUMN_NAME);
```

3、修改字段

```
ALTER TABLE TABLE_NAME MODIFY (COLUMN_NAME TYPE);
```

# 1.基本结构

## 1.1单/多查询

### 1.单查询

#### ● 1.1关于重复语句

```
SELECT DISTINCT DEPT_NAME FROM INSTRUCTOR;  
  
SELECT DEPT_NAME FROM INSTRUCTOR;
```

默认是保留重复元素,若要显示所有非重复元素则 **distinct**

```
SELECT ALL DEPT_NAME FROM INSTRUCTOR;
```

显示的指明保留重复语句

#### ● 1.2查询运算

```
SELECT ID, NAME, DEPT_NAME, SALARY*1.1  
FROM INSTRUCTOR;
```

将查询结果带入运算

### 2.多查询

```
SELECT NAME, COURSE_ID  
FROM INSTRUCTOR, TEACHES  
WHERE INSTRUCTOR.ID=TEACHES.ID;
```

多查询,用谓语句条件限制 \*笛卡尔积\*的无效性

查询结果仅仅是那些授课的导师



## 1.2 附加运算

### 1 更名运算

```
SELECT NAME AS INSTRUCTOR_NAME, COURSE_ID
FROM INSTRUCTOR, TEACHES
WHERE INSTRUCTOR.ID=TEACHES.ID;
```

与上面无异，只是多了更名运算，将name改名为instructor\_name，辨识度更高，从instructor中获取的name嘛

### 2 字符串运算

#### ● 2.1 大小写类运算

```
SELECT UPPER(NAME) AS INSTRUCTOR_NAME, LOWER(COURSE_ID)
FROM INSTRUCTOR, TEACHES
WHERE INSTRUCTOR.ID=TEACHES.ID;
```

与上面结果无异，但对结果的显示进行大小写转换

去除显示字符串后面的空格 trim()

#### ● 2.2 模式匹配

%（百分号）匹配任意多字符

\_（下划线）匹配一个字符

```
SELECT DEPT_NAME
FROM DEPARTMENT
WHERE BUILDING LIKE '%Watson%';
```

匹配building名称中含有 *Watson* 的

特殊匹配：转移字符

like 'ab/%cd%'escape '/'; 匹配以ab%cd开头的所有字符串

## ● 2.3所有属性

\*号查询

```
SELECT INSTRUCTOR.*
FROM INSTRUCTOR,TEACHES
WHERE INSTRUCTOR.ID=TEACHES.ID;
```

对一个笛卡尔积进行查询，可以看到 笛卡尔积的运算 结果

ID	NAME	DEPT_NAME	SALARY
▶ 10101	Srinivasan	Comp. Sci.	65000
10101	Srinivasan	Comp. Sci.	65000
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000

## ● 2.4排序

```
SELECT NAME
FROM INSTRUCTOR
WHERE DEPT_NAME='Physics'
ORDER BY NAME;
```

按照名字的字典序进行排序

默认是升序排列 **asc**  
若要降序排列 **desc**

```
SELECT *
FROM INSTRUCTOR
ORDER BY SALARY DESC, NAME ASC;
```

两个排序优先级，前者优先级更高，遇见相同的情况下按照第二优先级进行排序

## ● 2.5 where子句谓语句

对条件增加限定条件

**between** 范围限定 大于等于到小于等于

```
SELECT NAME
FROM INSTRUCTOR
WHERE SALARY BETWEEN 90000 AND 100000;
```

9万到10万之间

限定查询范围

等价语句：

```
SELECT NAME
FROM INSTRUCTOR
WHERE SALARY <= 100000 AND SALARY >= 90000;
```

<!--

部分SQL语句当两个元组的属性相同时的查询可以同类等价替换

```
SELECT NAME, COURSE_ID
FROM INSTRUCTOR, TEACHES
WHERE INSTRUCTOR.ID = TEACHES.ID AND DEPT_NAME = 'Biology';
```

上述语句oracle可正常查询

下列为替换语句，在oracle中无法正常运行



```
SELECT NAME, COURSE_ID
FROM INSTRUCTOR, TEACHES
WHERE (INSTRUCTOR.ID, DEPT_NAME) = (TEACHES.ID, 'Biology');
```

ORA-00920: 无效的关系运算符

-->

## 1.3 集合运算

基本的集合关系有 交 并 差 分别对应 **union intersect except**

交运算



```
(SELECT COURSE_ID
FROM SECTION
WHERE SEMESTER='Fall' AND YEAR =2017)
UNION
(SELECT COURSE_ID
FROM SECTION
WHERE SEMESTER='Spring' AND YEAR=2018);
```

相当于对两个运算的结果进行并集处理罢了

交运算



```
(SELECT COURSE_ID
FROM SECTION
WHERE SEMESTER='Fall' AND YEAR =2017)
INTERSECT
(SELECT COURSE_ID
FROM SECTION
WHERE SEMESTER='Spring' AND YEAR=2018);
```

相当于对两个结果进行交运算罢了

## 差运算

```
(SELECT COURSE_ID
FROM SECTION
WHERE SEMESTER='Fall' AND YEAR =2017)
MINUS
(SELECT COURSE_ID
FROM SECTION
WHERE SEMESTER='Spring' AND YEAR=2018);
```

左边减去右边 oracle用的是 minus 其他数据库用的或者是 except

在其他运算中 如果想要重复项也加入 差运算 用 **except all**

对比则oracle用的是 multiset except

差运算可以利用想象韦恩图，在数据库中就是返回那些出现在第一个表但是没有出现在第二个表的数据

## 1.4空值

主要是关于空值运算 null , unknown

由于不知道空值代表什么

1<null 可以为false或者true，故 这是没有意义的

### 关于那三张常见的集合运算

and 有假则假 true and unknown的结果为 unknown false and unknown的结果为false

false and false 的结果为false , unknown and unknown的结果为unknown

or 有真则真 true or unknown -> true

false or unknown -> unknown

unknown or unknown ->unknown

not 取反

### 空值查找

```
SELECT NAME
FROM INSTRUCTOR
WHERE SALARY IS NULL;
```

is unknown / is not unknown查找

```
SELECT NAME
FROM INSTRUCTOR
WHERE SALARY > 10000 IS UNKNOWN;
```

oracle 似乎不支持 is unknown型的sql语句

## 1.5 聚集函数

基本聚集函数:

**avg**:平均值

**min**:最小值

**max**:最大值

**sum**:总和

**count**:计数

sum和avg必须作用在数字集上,其他可以作用在比如字符串上

既然是函数 那么表达形似是 比如 avg(example)

### 1 基本聚集的使用

求平均值的举例

```
SELECT AVG(SALARY)
FROM INSTRUCTOR
WHERE DEPT_NAME='Comp. Sci.';
```

以为是自己的机器性能太差了,实则Comp. 的点号后面有空格,关于这种名字还是复制粘贴比较好

上述换名



```
SELECT AVG(SALARY) as AVG_SALARY
FROM INSTRUCTOR
WHERE DEPT_NAME='Comp. Sci.';
```

换名的好处是看着直观点

统计 count



```
SELECT COUNT(DISTINCT ID)
FROM TEACHES
WHERE SEMESTER='spring' and YEAR=2018;
```

统计 且去重 "在2018年春季授课的老师人数"

广泛查找



```
SELECT COUNT(*)
FROM COURSE;
```

## 2分组聚集

将有相同属性值的一些元组进行一起操作计算



```
SELECT AVG(SALARY)
FROM INSTRUCTOR;
```

广泛计算

下面为按照 系 进行划分

```
SELECT DEPT_NAME,AVG(SALARY)
FROM INSTRUCTOR
GROUP BY DEPT_NAME;
```

信息	摘要	结果 1
DEPT_NAME		AVG(SALARY)
▶ Elec. Eng.		80000
Physics		91000
Comp. Sci.		3333333333333333
Finance		85000
Biology		72000
Music		40000
History		61000

下列为分组 **group by** 的有效性 应用

```
SELECT DEPT_NAME,COUNT(DISTINCT INSTRUCTOR.ID) AS INSTR_COUNT
FROM INSTRUCTOR,TEACHES
WHERE INSTRUCTOR.ID =TEACHES.ID
      AND SEMESTER='Spring'
      AND YEAR=2018
GROUP BY DEPT_NAME;
```

查询每个系在2018年春季的授课老师的人数:总和恰好为6个,与之前人数的查询结果一致

信息	摘要	结果 1
DEPT_NAME		INSTR_COUNT
▶ Comp. Sci.		3
Finance		1
History		1
Music		1

关于上述的查询的语法逻辑可以好好学学



### 3 having 语句

```
SELECT DEPT_NAME,AVG(SALARY) AS avg_salary
FROM INSTRUCTOR
GROUP BY DEPT_NAME
having avg(SALARY)>42000;
```

分组统计

按照 系分组,同时求 平均工资高于42000的系的平均工资

<!--

与select语句类似,任何出现在having中,但是没有被聚集的属性必须出现在group by子句中???

-->

having的复杂查询实例

```
SELECT COURSE_ID, SEMESTER, YEAR, SEC_ID,AVG(TOT_CRED)
FROM STUDENT,TAKES
WHERE STUDENT.ID=TAKES.ID
      AND YEAR=2017
GROUP BY COURSE_ID, SEMESTER, YEAR, SEC_ID
HAVING COUNT(STUDENT.ID)>=2;
```

COURSE\_ID,SEMESTER,YEAR,SEC\_ID将这4个属性相同的元组分为一类,统计该类的学生的总学分的平均值

关于在group by语句中having和where这类逻辑谓语句之间的关系:

where用在分组前(整表), having用在分组之后(子表)

但是两者都是优化查询的结构谓语句

注: 关于 长/复杂 逻辑的查询语句类的执行顺序

1--> from

2--> where

3-->group by

4--> having

5--> select

题外话：关于复杂语句书写的技巧：

引入一种调试思维（分步思维）采取一步一步查询的方式，如：在两个表的嵌套查询中：可以先将两个表打印出来查看信息：找到所需属性再按需操作（结果1.结果2，结果3.....）

## 4对空值和布尔函数的聚集

```
SELECT SUM(SALARY)
FROM INSTRUCTOR;
```

在这个求和语句中,值为空 null的情况被忽略, 最终结果并不是null

### 1.6嵌套子查询

稍微复杂点的综合性查询(在逻辑上稍微复杂点)

## 1集合成员资格

IN 和 NOT IN的应用

```
(SELECT COURSE_ID
FROM SECTION
WHERE SEMESTER='Spring' AND YEAR=2018);
```

简单查询 将上一步看作是二次查询的查询范围

```
SELECT DISTINCT COURSE_ID
FROM SECTION
WHERE SEMESTER='Fall' AND YEAR=2017
      AND COURSE_ID IN(SELECT COURSE_ID
                        FROM SECTION
                        WHERE SEMESTER='Spring' AND YEAR=2018);
```

查询结果为那些 在 2017年秋季开设的课程同时也在2018年春季开设的课程

```
SELECT DISTINCT COURSE_ID
FROM SECTION
WHERE SEMESTER='Fall' AND YEAR=2017
      AND COURSE_ID NOT IN(SELECT COURSE_ID
                           FROM SECTION
                           WHERE SEMESTER='Spring' AND YEAR=2018);
```

查询结果为那些 不在 2017年秋季开设的课程同时也在2018年春季开设的课程

IN 和 NOT IN 应用于 枚举 中

```
SELECT DISTINCT NAME
FROM INSTRUCTOR
WHERE NAME NOT IN('Mozari','Einstein');
```

枚举排除了这两个人

下列为单属性中测试成员资格

下面这里应该是错的：准确来说IN是应用于精确匹配的

```
SELECT COUNT(DISTINCT ID)
FROM TAKES
WHERE (COURSE_ID, SEC_ID, SEMESTER, YEAR)
IN
(SELECT COURSE_ID, SEC_ID, SEMESTER, YEAR
FROM TEACHES
WHERE TEACHES.ID='10101');
```

注：COURSE\_ID, SEC\_ID, SEMESTER, YEAR是takes的主键，也是一些其他表的主键：可以看作是通用的

即将：COURSE\_ID, SEC\_ID, SEMESTER, YEAR看着是某段课程的键码。

大学关系型数据库中takes是存放学生学分情况的

查询结果即为："选修了10101的教师所授课的课程段的(不同)学生的总数".

还是自己的排版最好看捏~(￣▽￣)~\* SQL美化工具不靠谱

感觉完全熟练了解大学关系模型每个表的具体含义和内容还是非常有用的

## 2集合比较


查询在所有教师中 工资至少比Biology系中某个教师工资要高的所有教师的姓名

```
SELECT DISTINCT NAME
FROM INSTRUCTOR T, INSTRUCTOR S
WHERE T.SALARY > S.SALARY AND S.DEPT_NAME = 'Biology';
```

Oracle似乎并不支持该类语句的查询???

NO oracle只是对表的改名后面不使用AS 罢了

下列为SQL提供的另一种查询方式



```
SELECT NAME
FROM INSTRUCTOR
WHERE SALARY
> SOME
(SELECT SALARY
FROM INSTRUCTOR
WHERE DEPT_NAME='Biology');
```

但是SQL支持上述这种语句（将这种复杂查询分步进行再整合可以简化书写/理解难度）


关于**some**

SQL支持 >some <some =some这类语句

=some本质上就是 in

然后<>some并不等价于not in（显然<>some范围更加宽广）not in反而看着是将范围缩小了

下列展示 **all**运算



```
SELECT NAME
FROM INSTRUCTOR
WHERE SALARY
> ALL
(SELECT AVG(SALARY)
FROM INSTRUCTOR
GROUP BY DEPT_NAME);
```

查询结果为“工资比所有系的平均工资要高的教师的名字”

> all “比所有的都大”

下列为另一个同类例子：



```

SELECT AVG(SALARY)
FROM INSTRUCTOR
GROUP BY DEPT_NAME;
-- 得到每个系的平均工资

SELECT DEPT_NAME
FROM INSTRUCTOR
GROUP BY DEPT_NAME
HAVING AVG(SALARY)
>=ALL
(SELECT AVG(SALARY)
FROM INSTRUCTOR
GROUP BY DEPT_NAME);
-- 查找平均工资高于其他所有系的系名，（既是工资最高的系）

```

该例子中我才用分步的方式：使得查询逻辑得到简化

其中 **>=all** 可以用max聚集函数简化

### 3空关系测试

空关系测试 既 **exists** 或者 **not exists**

示例：

```

SELECT COURSE_ID
FROM SECTION A
WHERE SEMESTER='Fall' AND YEAR=2017
AND EXISTS
(SELECT *
FROM SECTION B
WHERE SEMESTER='Spring' AND YEAR=2018 AND A.COURSE_ID=B.COURSE_ID);
-- Oracle 并不太喜欢换名

```

可以用集合运算等价替换这个示例

EXISTS用于选择那些子查询中有返回任何一行数据的那些对应元组

下列示范not exists

```

SELECT COURSE_ID
FROM SECTION
WHERE SEMESTER='Fall' AND YEAR=2017
AND NOT EXISTS
(SELECT *
FROM SECTION
WHERE SEMESTER='Spring' AND YEAR=2018 AND
SECTION.COURSE_ID=SECTION.COURSE_ID);

```

两个查询语句没有太大区别

NOT EXISTS用于选择那些子查询没用返回任何一行数据的那些对应元组

下列示范**except**

似乎Oracle并不支持except

下列为 关于 “找出选修了ID为10101的教师所授课程段的（不同）学生的总数 的 exists写法

```

SELECT COUNT(DISTINCT ID)
FROM TAKES
WHERE EXISTS
(SELECT COURSE_ID, SEC_ID, SEMESTER, YEAR
FROM TEACHES
WHERE TEACHES.ID='10101'
AND TAKES.COURSE_ID=TEACHES.COURSE_ID
AND TAKES.SEC_ID=TEACHES.SEC_ID
AND TAKES.SEMESTER=TEACHES.SEMESTER
AND TAKES.YEAR=TEACHES.YEAR);

```

实测删除某个 = 关系得到的结果也是6 只要其他的 = 关系能够实现确定某个学生结果就可以使正确的，考虑到其4个属性都是主键：有些数据库可能会发生错误

## 4重复元组存在性测试

主要是关于**unique**语句

```
SELECT COURSE.COURSE_ID
FROM COURSE
WHERE UNIQUE
(SELECT SECTION.COURSE_ID
FROM SECTION
WHERE COURSE.COURSE_ID=SECTION.COURSE_ID
AND SECTION.YEAR=2017);
```

由于unique尚未广泛使用：故认为Oracle未实现该语句，实际结果也是报错

Oracle中的unique似乎不是怎么用的???

下列语句同样无法实现

```
SELECT COURSE.COURSE_ID
FROM COURSE
WHERE 1>=
(SELECT SECTION.COURSE_ID
FROM SECTION
WHERE COURSE.COURSE_ID=SECTION.COURSE_ID
AND SECTION.YEAR=2017);
```

unique既是 仅仅出现一次或者0次，同类语句 not unique（既最少出现两次）就不再赘述



## 5from语句中的嵌套子查询

```
SELECT DEPT_NAME, avg_salary
FROM (SELECT DEPT_NAME ,AVG(SALARY) AS avg_salary
      FROM INSTRUCTOR
      GROUP BY DEPT_NAME)
WHERE avg_salary>42000;
```

查询结果为 “系平均工资超过42000的那些系的平均工资”

下列为对结果关系的 命名化处理

```
SELECT MAX(tot_salary)
FROM (SELECT DEPT_NAME,SUM(SALARY)
      FROM INSTRUCTOR
      GROUP BY DEPT_NAME)AS dept_total(DEPT_NAME,tot_salary);
```

嵌套语句挺不错的，逻辑清晰，但是好像Oracle并不支持？？

## 6with子句

with子句提供了一种临时的关系表：离开该语句就停止生效了

例子：找到具有最大预算值的那些系

```
WITH max_budget (VALUE) AS (SELECT MAX(BUDGET) FROM DEPARTMENT)
SELECT BUDGET
FROM DEPARTMENT,
     max_budget
WHERE DEPARTMENT.BUDGET = max_budget.VALUE;
```

可见 **with** 提供了一种单值单属性的关系表，但是似乎Oracle好像又不支持该语句？X 注意 navicat的补全是一些特定的关键字导致错误，不一定是Oracle不支持该类语法，请注意辨析

注：DataGrip的SQL美化好像看着确实舒服

例2 “找出工资总额大于所有系平均工资总额的所有系”

就是找出哪些工资总额高于 所有系平均总额的系 即可

```
WITH DEPT_TOTAL (DEPT_NAME, value) AS (  
    SELECT DEPT_NAME, sum(SALARY)  
    FROM INSTRUCTOR  
    GROUP BY DEPT_NAME  
) ,    --定义系和总值表  
DEPT_TOTAL_AVG AS (  
    SELECT avg(value) AS avg_value  
    FROM DEPT_TOTAL    --定义平均值表(单值表-所有系的总值的平均值)  
)  
SELECT DEPT_NAME  
FROM DEPT_TOTAL, DEPT_TOTAL_AVG  
WHERE value > DEPT_TOTAL_AVG.avg_value;
```

像这种复杂的嵌套查询，注意其中的关键字，大小写之类的

## 7标量子查询

查询只返回同一个包含单个属性的元组的子查询称为标量子查询

上面的实例中似乎包含一个标量子查询

```
SELECT DEPT_NAME,  
    (SELECT COUNT(*)  
     FROM INSTRUCTOR  
     WHERE DEPARTMENT.DEPT_NAME = INSTRUCTOR.DEPT_NAME)  
     AS num_instructors  
FROM DEPARTMENT;  
  
SELECT e.first_name,  
       e.salary,
```

```
(SELECT MAX(salary) FROM employees WHERE department_id =  
e.department_id) as max_salary  
FROM employees e;
```

上面的查询将返回每个员工的名字、薪水以及他们所在部门的最高薪水。

查询结果是： 找出所有的系，以及每个系有多少名教师

将语法拆开不难看懂，而且确实很灵活方便

## 8不带from的子句查询

部分数据库系统严格按照 select-where-from 执行，那么不带from子句在部分数据系统中可能报错

例子 "平均每位教师所授课数"

```
(SELECT COUNT(*) FROM TEACHES)  
/  
(SELECT COUNT(*) FROM INSTRUCTOR);
```

对于部分不支持去掉from的数据库系统 使用DUAL虚拟关系

```
SELECT(SELECT COUNT(*) FROM TEACHES)  
/  
(SELECT COUNT(*) FROM INSTRUCTOR)  
FROM DUAL;
```

似乎Oracle不支持上述虚拟关系？

但是下列语句在Oracle中又可以运行

```
SELECT (SELECT COUNT(*) FROM TEACHES) / (SELECT COUNT(*) FROM  
INSTRUCTOR) AS EXANPLE  
FROM DUAL;
```

注：第一个的结果里面是 15 恰好是一个整型数字，第二个结果里是一串更长的数字，其原因是为什么呢？

解释（来自bing AI）：

前者是一个单独的计算式子，整数不会被截断，两个整数做除法，结果转为了一致的整数，后者除法运算发生在select语句中，整数的除法运算被截断为浮点数，最终的计算结果是浮点数

## 1.7数据库数据的修改

### 1 删除

格式

```
delete from r  
where P;
```

删除所有元组

```
delete from r;
```

原理是因为没有 WHERE谓语句进行具体的筛选

例1



```
delete from instructor
where dept_name='Finance'
```

例2



```
delete from instructor
where salary between 12000 and 15000;
```

删除语句结合where的语法罢了

例三 嵌套关系的删除 也就是一种嵌套关系罢了



```
DELETE FROM INSTRUCTOR
WHERE DEPT_NAME
IN (
SELECT DEPT_NAME
FROM DEPARTMENT
WHERE BUILDING = 'Waston');
```

删除哪些在Waston教学楼的系

例四 删除哪些工资低于平均工资的老师



```
DELETE FROM INSTRUCTOR
WHERE SALARY < (SELECT AVG(SALARY) FROM INSTRUCTOR);
```

## 2插入（增加）

例一



```
INSERT INTO COURSE
VALUES('CS-437','Database Systems','Comp. Sci.',4);
```

基础的插入语句

例二



```
INSERT INTO COURSE(TITLE,COURSE_ID,CREDITS,DEPT_NAME)
VALUES('CS-437','Database Systems','Comp. Sci.',4);
```

自定义插入值属性的顺序

例三 按照查询结果插入系



```
INSERT INTO INSTRUCTOR
SELECT ID,NAME,DEPT_NAME,18000
FROM STUDENT
WHERE DEPT_NAME='Music'AND TOT_CRED>144;
```

插入查询的返回结果

关于主码约束上的插入细节



```
INSERT INTO STUDENT
SELECT *
FROM STUDENT;
```

如果没有主键约束，该语句会导致在STUDENT中插入无数条语句

### 3更新

例一

```
UPDATE INSTRUCTOR
SET SALARY=SALARY*1.05;
```

将工资提升1.05倍

例二

```
UPDATE INSTRUCTOR
SET SALARY=SALARY*1.05
WHERE SALARY<(SELECT avg(SALARY)
               FROM INSTRUCTOR);
```

按照查询结果更新数据

例三

```
UPDATE INSTRUCTOR
SET SALARY=SALARY*1.05
WHERE SALARY<=100000;

UPDATE INSTRUCTOR
SET SALARY=SALARY*1.03
WHERE SALARY>100000;
```

根据情况的两次更新，但是会出现幸运儿低于100000一点点先被\*1.05 之后又被 \* 1.03导致错误

解决方案是：

1，将更新顺序正确书写

2, 将两条更新语句放在一条更新语句中

下面是条件分支语句 的 **CASE** 语句

```
UPDATE INSTRUCTOR
SET SALARY=CASE
    WHEN SALARY<=100000 THEN
        SALARY*1.05
    ELSE
        SALARY*1.03
END;
```

格式

```
case
    when p1 then result1
    when p2 then result2
    when p3 then result3
    . . . . .
end
```

下例为 标量子查询 在更新语句中的应用： 复杂的语句书写

```
UPDATE STUDENT
SET TOT_CRED=(
    SELECT SUM(CREDITS)
    FROM TAKES,COURSE
    WHERE STUDENT.ID=TAKES.ID
        AND TAKES.COURSE_ID=COURSE.COURSE_ID
        AND TAKES.GRADE<>'F'
        AND TAKES.GRADE IS NOT NULL);
```

将每个student元组中的tot\_cred属性值设置为该生成功学完的课程学分总和。



下面为同义替换语句

```
UPDATE STUDENT
SET TOT_CRED = (SELECT CASE
                  WHEN SUM(CREDITS) IS NOT NULL THEN
                      SUM(CREDITS)
                  ELSE
                      0
                  END
                FROM TAKES,
                     COURSE);
```

似乎化繁为简 但是 可读性降低了不少

## 2. 中级SQL

简述：更复杂形式的SQL查询，视图定义，事务，以及完整性约束

### 1. 连接表达式

目的：用“连接”运算（自然连接）来表达某些笛卡尔积难以表达的查询

#### 1.1 自然连接

注：navicat的快捷键：ctrl+e 选中一个SQL代码段落 段落！再ctrl+r 就可以快捷运行选中代码

ctrl+q 开启一个全新的查询（新建查询）

**自然连接**：作用于两个关系，并且产生/返回一个关系，自然连接只考虑都出现的那些属性上取值相同的元组（共同属性—>取值相同的元组）—>返回新的行

```
SELECT *
FROM STUDENT NATURAL JOIN TAKES;
```

典型的自然连接实例 ID是上述两个关系的共同属性

结果如下：（部分省略）

ID	NAME	DEPT_NAME	TOT_CRED	COURSE_ID	SEC_ID	SEMESTER
----	------	-----------	----------	-----------	--------	----------

出现的列的顺序如下：

共同属性>>前关系表的属性>>后关系表的属性

之前不知道 笛卡尔积 的连接结果产生了副本,该连接方式类似于表的拓展,(更为自然的连接)

## 实例2

```
-- 笛卡尔积查询
SELECT NAME,COURSE_ID
FROM STUDENT,TAKES
WHERE STUDENT.ID=takes.ID;
-- 符号对称的笛卡尔积查询
SELECT NAME,COURSE_ID
FROM STUDENT CROSS JOIN TAKES
WHERE STUDENT.ID=takes.ID;

-- 自然连接
SELECT NAME,COURSE_ID
FROM STUDENT NATURAL JOIN TAKES;

-- 上述三者查询结果一致
```

多表的自然连接查询

```
SELECT A1,A2,.....
FROM R1 NATURAL JOIN R2
      NATURAL JOIN R3
      .....
WHERE P1,.....;
```

```
SELECT NAME,TITLE  (*)是下面图片所使用的查询结果
FROM STUDENT NATURAL JOIN TAKES,COURSE
WHERE TAKES.COURSE_ID=COURSE.COURSE_ID;
-- 先是自然连接返回一张新表 再与后面的 COURSE进行 笛卡尔积连接
```

```
-- 下面查询的结果与上面一致
SELECT NAME,TITLE
FROM STUDENT NATURAL JOIN TAKES
      NATURAL JOIN COURSE;
-- 自然连接的自动属性匹配 但是不太适合多个属性相同的表关系中，会出现意想不到的效果
```

COURSE_ID	SEC_ID	SEMESTER	YEAR	GRADE	COURSE_ID(1)	TITLE
-----------	--------	----------	------	-------	--------------	-------

看这第一个查询应该就可以明确知道 含有 笛卡尔积 的查询结果

上述第二个查询会出现自然连接问题：

这两个查询的结果应该是相同的，前提是 `STUDENT`、`TAKES` 和 `COURSE` 表中没有其他名称相同的列。否则，`NATURAL JOIN` 可能会产生意料之外的结果。因此，通常建议明确指定连接条件，以避免可能的混淆和错误。（前一个结果的 `dname`，`courseid` 和后面表 `course` 都有这两个属性）

前者22条结果 后者17条结果

提示：typora使用技巧 飘号 内容 飘号 可以让内容高亮变红

关于自然连接问题的解决方案，手动 属性匹配

```
SELECT NAME,TITLE
FROM (STUDENT NATURAL JOIN TAKES) JOIN COURSE USING (COURSE_ID);
```

显示的指明了属性名列表，该SQL查询给出了正确的答案

注：关于这个问题可以上课看看那老师怎么讲的

大概是因为相同的属性名导致多种可行的自然连接方案，随机某种方案匹配？但总是不确定的方案进行匹配

稍微区分下 自然连接 `natural` 与 `on` 的使用的区别

## 1.2连接条件

```
SELECT *
FROM STUDENT JOIN TAKES ON STUDENT.ID=TAKES.ID;
-- 标准的SQL连接方式查询

SELECT *
FROM STUDENT,TAKES
WHERE STUDENT.ID=TAKES.ID;
-- 笛卡尔积式连接查询
-- 两者查询的结果一致 都产生了一个ID(1)副本
```

关于副本ID(1)表示如下：

ID	NAME	DEPT_NAME	TOT_CRED	ID(1)	COURSE_ID	SEC_ID
----	------	-----------	----------	-------	-----------	--------

在1.1和1.2之间的两种连接方式的区别（`using` 和 `on`）

**using**是再两张表有明确的相同属性时

**on**时指定两张表的各自的某个属性值

下面时只显示一个ID的版本

```
SELECT STUDENT.ID AS
ID,NAME,DEPT_NAME,TOT_CRED,COURSE_ID,SEC_ID,SEMESTER,YEAR,GRADE
FROM STUDENT JOIN TAKES ON STUDENT.ID=TAKES.ID;
--其中STUDENT.ID 可以替换成 TAKES.ID
```

这次的查询结果与STUDENT和TAKES两涨表的自然连接的结果一致咯

但是在SELECT处算是主动将重复的ID项去除

关于内连接和外连接的区别：

简述就是：

内连接和外连接的主要区别在于它们如何处理不匹配的行：内连接忽略不匹配的行，而外连接则包含不匹配的行并用NULL填充缺失的列

关于ON的出现

很多ON的语句都可以用WHERE谓语句进行处理，ON的主要作用是体现在外连接中

## 3外连接

```
SELECT *  
FROM STUDENT NATURAL JOIN TAKES;
```

为什么引入外连接：？

因为如上查询在某些同学并未选课的情况下是不会出现其对应的行的，那么与自己想要的结果是不一样的

所以引入外连接实现将没有匹配的行表示为NULL

为了保留那些会丢失的元组：

**左外连接**：只保留出现在左外连接运算之前（左边）的关系中的元组

**右外连接**：.....右边.....

**全外连接**：保留左右边

实例代码：

```
SELECT *  
FROM STUDENT NATURAL LEFT OUTER JOIN TAKES;
```

外连接的好处如下：

ID	NAME	DEPT_NAME	TOT_CRED	COURSE_ID	SEC_ID	SEMESTER
55739	Sanchez	Music	38	MU-199	1	Spring
70557	Snow	Physics	0	(Null)	(Null)	(Null)

外连接应用实例：

```
SELECT ID  
FROM STUDENT NATURAL LEFT OUTER JOIN TAKES  
WHERE COURSE_ID IS NULL;
```

信息	摘要	结果 1
ID		
▶ 70557		

找出一门课程也没选修的学生的ID

左外连接和右外连接具有对称性



```
SELECT *
FROM TAKES NATURAL RIGHT OUTER JOIN STUDENT;
```

结果与左外连接的结果几乎一致，但是属性的出现顺序不一样

下面展示全外连接：



```
SELECT *
FROM (SELECT *
      FROM STUDENT
      WHERE DEPT_NAME='Comp. Sci.')
      NATURAL FULL OUTER JOIN
      (SELECT *
      FROM TAKES
      WHERE SEMESTER='Spring' AND YEAR=2017);
```

结果如下：

信息	摘要	结果 1
ID	NAME	DEPT_NAME
▶ 12345	Shankar	Comp. Sci.
54321	Williams	Comp. Sci.
76653	(Null)	(Null)
76543	Brown	Comp. Sci.
00128	Zhang	Comp. Sci.

查询结果是：

"显示Comp. Sci. 系的所有学生，以及他们在2017年春季选修的所有课程段的列表"

简言之 就是可以看出计算机系的所有学生，和2017年春季的所有可程，以及他们是否选修了 这些课程

下面展示外连接配合ON使用

```
SELECT *  
FROM STUDENT LEFT OUTER JOIN TAKES ON STUDENT.ID=TAKES.ID;
```

用ON的主动限定条件就省略的NATURAL的自然匹配连接，但是结果中ID显示了两遍

下面是ON或者WHERE对外连接的不同表现

```
SELECT *  
FROM STUDENT LEFT OUTER JOIN TAKES ON TRUE  
WHERE STUDENT.ID=TAKES.ID;
```

上述代码在ORACLE中好像无法运行，但是结论是 TRUE 将每项空值都转为了TRUE，那么结论是结果自然不会出现NULL，而是直接略过那条元组，既是SNOW同学不会出现在结果中

## 4连接类型和条件

JOIN子句中不写OUTER时默认时内连接方式

自然连接等价于自然内连接

(NATURAL JOIN -> NATURAL INNER JOIN)

主要的分类就是：？

常规连接 ： 外连接

内连接 ： 外连接

## 2视图

## 3.高级SQL

### 1高级聚集函数

#### 1.1排名

##### ● rank()函数

```
SELECT ID,RANK() OVER (ORDER BY (GPA) desc) as s_rank
FROM STUDENT_GRADES;
```

`RANK() OVER (ORDER BY (GPA) desc) as s_rank` 是一个窗口函数，用于计算每个学生的排名。

使用rank()函数接入一组数据 desc告诉rank函数的数据处理方式

ORDER BY (GPA) desc表示为GPA的值进行降序给出排名值

可见窗口函数的概念和意义

```
SELECT ID,RANK() OVER(ORDER BY (GPA) DESC) AS S_RANK
FROM STUDENT_GRADES
ORDER BY S_RANK;
```

这里是对显示结果进行排序展示

下面是对空的GPA为null 把排序的排序的情况后面放

```
SELECT ID,RANK() OVER (ORDER BY (GPA) desc nulls last ) as s_rank
FROM STUDENT_GRADES;
```



## ● 分区 rank



```
SELECT ID,DEPT_NAME,RANK() OVER (PARTITION BY DEPT_NAME ORDER BY GPA
DESC) AS DEPT_RANK
FROM STUDENT_GRADES

ORDER BY DEPT_NAME,DEPT_RANK;
// 后面这里是指定显示的 排序顺序的
// PARTITION BY DEPT_NAME 指定排名分区的
```

也就是在 rank那个窗口中 指明 PARTITION BY DEPT\_NAME  
PARTITION

## ● 分组rank



```
SELECT ID,NTILE(4) OVER (ORDER BY (GPA DESC)) AS PARG_RANK
FROM STUDENT_GRADES;
```

NTILE(4) OVER (ORDER BY (GPA DESC))  
最后显示位于的排位分区 介于1到4之间

## 1.2分窗

实例1



```
SELECT YEAR,AVG(NUM_CREDIT) OVER (ORDER BY YEAR ROWS 3 PRECCDING) AS
AVG_TOTAL_CREDITS
FROM TOT_CREDITS;
```

查询结果是将 每个年份于前两年做一起进行分窗平均值处理  
对于2019年就是将2017年和2018年加一起计算平均值  
AVG(NUM\_CREDIT) OVER (ORDER BY YEAR ROWS 3 PRECCDING)  
对AVG函数进行分窗：按照年份排序取3年

包括当前行就是每次计算4行，前面1，2，3行可能计算不到4行，但是后面是这样的，如果需要计算后四行可以使用排序？

## 实例2



```
AVG(NUM_CREDIT) OVER (ORDER BY YEAR ROWS UNBOUNDED PRECCDING)
```

将具体的 3 改为UNBOUNDED则可得到 该年及之前的所有年份的平均值