# 分组数据

可以使用GROUP BY子句将查询结果分成若干组。

## 查询语法

**SELECT column, group\_function(column)**

**FROM table**

**[WHERE condition]**

**[GROUP BY group\_by\_expression]**

**[ORDER BY column];**

执行过程：

FROM ->

WHERE,行筛选 ->

GROUP BY,分组 ->

SELECT,针对每组得到统计结果，每组得到一个结果 ->

ORDER BY

注意：

1. GROUP BY在WHERE之后。
2. 分组的列，可以在统计时，查询出来。

一组中分组的列值相等的，查询出来更有意义。

1. WHERE中不能直接使用分组函数，此时还没有进行分组。
2. 按照多个列分组，多个列值都相等算作一组。

## 通过例子学习分组数据

例1：统计每个部门的人数。

|  |
| --- |
| SELECT dept\_id,COUNT(id)  FROM s\_emp  GROUP BY dept\_id; |

例2：统计每个销售人员对应的客户数量。

分析：把所有的客户都列出来，然后按照客户对应的销售分组。这样就可以计算每个销售对应的客户数量了。

|  |
| --- |
| SELECT sales\_rep\_id,COUNT(id)  FROM s\_customer  GROUP BY sales\_rep\_id; |

例3：统计每个区域的部门的数量。

|  |
| --- |
| SELECT region\_id,COUNT(id)  FROM s\_dept  GROUP BY region\_id; |

例4：统计1号区域每个部门的人数。

分析：最终计算的是人数，所以需要查询s\_emp(员工表)，是把查询得到的员工集按照分组进行统计。

最终显示的是部门编号，已经统计得到的这个部门的人数。需要查询s\_dept，因为指定了1号区所在的部门。

|  |
| --- |
| SELECT e.dept\_id, count(e.dept\_id)  FROM s\_emp e, s\_dept d  WHERE e.dept\_id=d.id and d.region\_id=1  GROUP BY e.dept\_id; |

例5：统计每个区域员工的数量，平均薪资。

分析：统计得是员工的数量和平均薪资，这些需要查s\_emp;

最后是根据区域分组，要把每个区域进行统计，需要查s\_region。

从s\_emp不能直接关联到s\_region，必须经过s\_dept；

最后需要显示的字段，一个是区域id，这个数据来自于s\_region。

统计是按照区域进行的，所有需按照区域id进行分组。

|  |
| --- |
| SELECT r.id, count(r.id), avg(e.salary)  FROM s\_emp e, s\_dept d, s\_region r  WHERE e.dept\_id=d.id and d.region\_id=r.id  GROUP BY r.id; |

例6：统计每个部门每个职位的人数。

分析：统计信息来自员工表，s\_emp，表中有统计的部门信息(dept\_id)，职位信息(title)。

根据要求，最后的统计是依据分组部门+职位。

|  |
| --- |
| SELECT dept\_id, title, count(id)  FROM s\_emp  GROUP BY dept\_id, title; |

例6：统计每个部门的人数,显示部门编号，部门名，部门人数。

分析：查询的表应该有s\_emp和s\_dept。

两者的关联是：s\_emp.dept\_id=s\_dept.id。

最后要统计得是部门人数，要把员工表按照部门分组，每组的计数就是部门人数了。

要注意：最后需要显示的是三个字段，其中部门名要出现在select中，这样也必须出现在group by的子句中。

而在GROUP BY中增加部门名，会影响最后的查询结果吗？这里是不会的，因为部门id相同，部门名字肯定也相同。

|  |
| --- |
| SELECT e.dept\_id,d.name,COUNT(e.dept\_id)  FROM s\_emp e,s\_dept d  WHERE e.dept\_id=d.id  GROUP BY e.dept\_id,d.name; |

注意：出现在select中的字段（不是组函数），一定要在group by子句中，否则会报错: select不是GROUP BY表达式。

## HAVING子句 – 过滤分组数据

Where和Having都是对查询结果的一种筛选，也就是设定查询条件的语句。

1).where 不能放在GROUP BY 后面。

2).HAVING 是跟GROUP BY 连在一起用的，放在GROUP BY 后面，此时的作用相当于WHERE。

3).WHERE 后面的条件中不能有聚集函数，比如SUM(),AVG()等，而HAVING 可以。

**查询语法：**

SELECT ……

FROM ……

WHERE 条件

GROUP BY column

HAVING 条件

ORDER BY column;

**执行流程**

FROM ->

WHERE，行筛选 ->

GROUP BY，分组 ->

HAVING,组筛选 ->

SELECT,统计 ->

ORDER BY

**注意：**

HAVING,使用前提：GROUP BY，分组。

作用：多分组后的多个组进行筛选

根据分组的列筛选，或者统计结果筛选

不能通过普通的列直接筛选，一组中普通的列值各不相等

语法同WHERE

例7：查询出部门人数在2人及2人以上的部门编号，人数，平均薪资。

|  |
| --- |
| SELECT dept\_id,COUNT(id),AVG(salary)  FROM s\_emp  GROUP BY dept\_id  HAVING COUNT(id)>=2; |

例8：查询部门平均薪资高于1000的部门编号，部门平均薪资。

|  |
| --- |
| SELECT dept\_id,AVG(salary)  FROM s\_emp  GROUP BY dept\_id  HAVING AVG(salary)>1000; |

查询出每个员工薪资都高于1000的部门编号，部门平均薪资，人数。

|  |
| --- |
| SELECT dept\_id,AVG(salary),COUNT(id),MIN(salary)  FROM s\_emp  GROUP BY dept\_id  HAVING MIN(salary)>1000; |

例9：查询出部门编号高于40的部门平均薪资，人数。

|  |
| --- |
| SELECT dept\_id,AVG(salary),COUNT(id)  FROM s\_emp  WHERE dept\_id>40  GROUP BY dept\_id;  或者：  SELECT dept\_id,AVG(salary),COUNT(id)  FROM s\_emp  GROUP BY dept\_id  HAVING dept\_id>40;  推荐第一种。 |

# 子查询 – 查询嵌套

查询中嵌套查询语句，把内部嵌套的查询称为子查询。

子查询可以出现的位置：

WHERE子句

HAVING子句

FROM子句

注意：查询中的子查询需要用()括起来。

## WHERE子查询

例1：查询薪资高于Ben(first\_name)的员工。

分析：查询的表，s\_emp，查询条件，是员工的薪水高于Ben的薪水。

所以，关键是要先知道Ben的薪水。用子查询查询出Ben的薪水。

查询Ben的薪资：

SELECT salary

FROM s\_emp

WHERE first\_name='Ben';

|  |
| --- |
| SELECT id,first\_name,salary  FROM s\_emp  WHERE salary>(  SELECT salary  FROM s\_emp  WHERE first\_name='Ben'  ); |

上例中，子查询出现在WHERE，是WHERE子查询。

例2：查询高于平均薪资的员工信息。

分析：先采用子查询查平均薪资，然后在查薪资高于1)结果的员工信息。

|  |
| --- |
| SELECT id,first\_name,salary  FROM s\_emp  WHERE salary>(  SELECT AVG(salary)  FROM s\_emp  ); |

例3：查询Elena(first\_name)所在部门所有员工的平均薪资。

分析：先查Elena所在的部门，得到部门的id，然后查属于这个部门所有的员工的平均薪资。

|  |
| --- |
| SELECT avg(salary)  FROM s\_emp  WHERE dept\_id = (  SELECT dept\_id  FROM s\_emp  WHERE first\_name='Elena'  ); |

例4：查询1号区域所有员工的平均薪资(通过子查询来完成)。

分析：先查询1号区域内的所有的部门，然后查这些部门的所有员工。

查询1号区域内的部门，结果不止一个，产生了多行的结果。需要用IN来判断。

|  |
| --- |
| SELECT AVG(salary)  FROM s\_emp  WHERE dept\_id IN(  SELECT id  FROM s\_dept  WHERE region\_id=1  ); |

单行子查询结果 ，可以判断是否等于一个结果。

多行子查询结果 ，可以用IN判断是否在其中。

比子查询所有值都大=>比最大值还大，max()

比子查询中某一个值大=>比最小值还大,min()

>ALL(子查询) 大于所有

>ANY(子查询) 大于某一个

<ALL(子查询) 小于所有

<ANY(子查询) 小于某一个

如果子查询结果返回单行多列，可以使用单行多列比较运算：

查找和Elena一个部门和一样职称(title)的所有员工。

|  |
| --- |
| SELECT dept\_id, title, id, first\_name, salary, start\_date  FROM s\_emp  WHERE (dept\_id, title)=(  SELECT dept\_id, title  FROM s\_emp  WHERE first\_name='Elena'  ); |

如果子查询结果返回的是多行多列，只能使用多行多列比较运算符（IN, ALL, ANY）。

另外要注意，多行子查询结果中的null问题：

查询不是老板的员工：

|  |
| --- |
| SELECT id, first\_name  FROM s\_emp  WHERE id NOT IN (  SELECT manager\_id  FROM s\_emp  WHERE manager\_id IS NOT null  ); |

## HAVING子查询

例5：查询人数高于44号部门人数的部门编号，部门人数。

分析：先查出44号部门的人数。然后，通过部门分组查找各部门的人数，并且过滤条件是这些部门的人数要大于44号部门的人数。

查找44号部门人数：

select count(id)

from s\_emp

where dept\_id=44;

|  |
| --- |
| SELECT dept\_id,COUNT(id)  FROM s\_emp  GROUP BY dept\_id  HAVING COUNT(id)>(  SELECT COUNT(id)  FROM s\_emp  WHERE dept\_id=44  ); |

上面的例子，子查询出现在HAVING子句，是HAVING子查询。

例6：查询部门平均薪资高于公司平均薪资的部门编号，部门平均薪资。

分析：公司平均工资

select avg(salary)

from s\_emp;

|  |
| --- |
| SELECT dept\_id,AVG(salary)  FROM s\_emp  GROUP BY dept\_id  HAVING AVG(salary)>(  SELECT AVG(salary)  FROM s\_emp  ); |

## FROM子查询

例7：查询部门平均薪资高于公司平均薪资的部门编号，部门名，部门平均薪资。

分析：这个例子比起上一个，只是在查询结果，多了一个部门名字。可以多查询一张部门表，关联后，分组按部门id和部门名字。

另外一个思路，可以把例6的查询结果，当做一个集合，然后和部门表管理，这样可以得到部门的名字。

|  |
| --- |
| SELECT e.dept\_id, d.name, AVG(e.salary)  FROM s\_emp e, s\_dept d  WHERE e.dept\_id=d.id  GROUP BY e.dept\_id, d.name  HAVING AVG(e.salary) > (  SELECT AVG(salary)  FROM s\_emp  ); |

|  |
| --- |
| SELECT t.dept\_id,d.name,t.avg\_sal  FROM (  SELECT dept\_id,AVG(salary) avg\_sal  FROM s\_emp  GROUP BY dept\_id  HAVING AVG(salary)>(  SELECT AVG(salary)  FROM s\_emp  )  ) t,s\_dept d  WHERE t.dept\_id=d.id; |

上面的例子中，子查询出现在了FROM子句，是FROM子查询。

## ROWNUM结合子查询

例8：查询员工表中的第一条到第五条记录,查询员工编号，员工名字，员工薪水。

|  |
| --- |
| SELECT id,first\_name,salary  FROM s\_emp  WHERE rownum<=5; |

例9：查询员工表第6~10条记录，查询员工编号，员工名字和薪水。

分析：前面已经讲过这样的查询，可以采用MINUS（差集）。这里的方式，结合子查询。

查询表的前10行：

SELECT id,first\_name,salary,rownum

FROM s\_emp

WHERE rownum<=10;

然后从这个结果集中查询第6行开始的行。

|  |
| --- |
| SELECT id, first\_name,salary  FROM(  SELECT id,first\_name,salary,rownum rn  FROM s\_emp  WHERE rownum<=10  )  WHERE rn>=6; |

例10：查询薪资最高的5个人的id，名字，薪资。

分析：先查询所有的员工薪资，按薪资的降序排序。然后从这个集合中查前5行即可。

|  |
| --- |
| SELECT id, first\_name, salary  FROM (  SELECT id,first\_name,salary  FROM s\_emp  ORDER BY salary DESC  )  WHERE rownum<=5; |

## 相关子查询

子查询执行过程中需要用到主查询的相关结果，即子查询在主查询返回的结果集上执行（针对主查询的每一行结果，子查询都要执行一次）。子查询和主查询在执行过程中相互依赖。

例11：查询员工薪资大于1000的所有员工编号，名字，薪资，部门编号，部门名字，并按薪资降序排序。

分析：因为查询结果有部门名字，所有可以关联员工表和部门表查询得到。

也可以采用子查询，单独查询每个部门的名字的方式，得到最后的查询结果。

|  |
| --- |
| SELECT id, first\_name, title, salary, dept\_id, (  SELECT s\_dept.name  FROM s\_dept  WHERE s\_dept.id=s\_emp.dept\_id) dept  FROM s\_emp  WHERE salary>1000; |

注意：主查询结果中的每一行，dept\_id，子查询都要相应的执行一次。

等价于

|  |
| --- |
| SELECT e.id, e.first\_name, e.title, e.salary, e.dept\_id, d.name dept  FROM s\_emp e, s\_dept d  WHERE e.dept\_id=d.id AND  e.salary>1000; |

在WHERE子句中使用关联子查询：

例12：查询员工：薪资高于同样职称(title)的员工的平均薪资,查询结果包含员工id，名字，职称和薪资，并按职称降序排列。

|  |
| --- |
| SELECT e1.id, e1.first\_name, e1.title, e1.salary  FROM s\_emp e1  WHERE e1.salary > (  SELECT avg(e2.salary)  FROM s\_emp e2  WHERE e2.title = e1.title  )  ORDER BY e1.title DESC; |

## 在WHERE中使用exists子查询

Exists子句的语法：

exists(SQL查询) -- 如果SQL查询有结果，exists为True，否则为False

not exists(SQL查询)

请看下面的查询语句：

SELECT id, first\_name

FROM s\_emp

WHERE exists (

SELECT \*

FROM s\_emp

WHERE dept\_id = 99

);

因为表s\_emp不存在dept\_id=99，所以这个子查询的结果是空，因此，exists返回false。这样，这个查询等价于：

SELECT id, first\_name

FROM s\_emp

WHERE 1=2;

同样的道理：

SELECT id, first\_name

FROM s\_emp

WHERE exists (

SELECT \*

FROM s\_emp

WHERE dept\_id = 10

);

由于有部门编号是10，就因此等价于：

SELECT id, first\_name

FROM s\_emp

WHERE 1=1;

例13：查询有员工的部门，显示部门id和部门名。

|  |
| --- |
| SELECT id, name  FROM s\_dept  WHERE exists  (SELECT id  FROM s\_emp  WHERE s\_dept.id = s\_emp.dept\_id); |

比较exists和in：

1. select \*

from s\_emp

where exists

(select id from s\_dept where s\_emp.dept\_id=s\_dept.id);

  s\_emp数据量小而s\_dept数据量非常大时，s\_emp<<s\_dept 时，这个查询效率高。

1. select \*

from s\_emp

where s\_emp.dept\_id in (select id from s\_dept);

  s\_emp数据量非常大而s\_dept数据量小时，s\_emp>>s\_dept 时，这样的查询效率高。

# DML语句—UPDATE和DELETE

所有的DML语句操作后：

**确认操作，提交：COMMIT;**

**取消操作，回滚：ROLLBACK;**

## UPDATE – 更新，修改

语法：

UPDATE 表名 SET 列=值,...,列=值

[WHERE 条件];

修改符合条件的行中对应的列值。

WHERE可以省略：表示修改所有的行。

例1：1号员工薪资修改为3000。

|  |
| --- |
| UPDATE s\_emp SET salary=3000  WHERE id=1; |

例2：1号员工的薪资增加500。

|  |
| --- |
| UPDATE s\_emp SET salary=salary+300  WHERE id=1; |

注意：

上面两个例子执行后，更新后的数据只是存在缓存中，并没有更新到数据库中。

如果执行了commit，确认，那么缓存的数据会更新到数据库中。

如果执行了rollback, 回滚，那么缓存的数据不会更新到数据库中，对数据库中的数据没有影响。

## DELETE - 删除

语法：

DELETE [FROM] 表名

[WHERE 条件];

删除符合条件的行。

如果WHERE省略，表示删除这个表中所有的行，清空。

例2：在部门表中新增一个部门，编号为99，部门名字为JAVA，地区编号1。确认新增后，再把新增的JAVA部门删除。

|  |
| --- |
| INSERT INTO s\_dept (id,name,region\_id)  VALUES (99,'JAVA',1);  COMMIT;  DELETE FROM s\_dept  WHERE id=99;  COMMIT; |

## 事务 – Transaction

假设A转账500元给B，这个过程中简单看包含：

A的余额减少500元。

B的余额增加500元。

以上两个过程。

如果1）成功了，但是2）失败了，那么系统中，A减少的500元“消失”了。

如果1）失败了，但是2）成功了，那么系统中，B增加的500元“来历不明”。

（在某些业务模型中，第二种情况是存在的，并不依赖于1）和2）的次序。）

显然，从转账这个“业务”看，1）和2）应该是一个整体，要么都成功，要么都失败。他们应该是一个整体。

把业务过程的操作看做一个整体，如果某个操作失败了，那么整个业务过程就“回滚”到业务开始之前的状态。只有所有的操作都成功了，确认后业务才会进行到业务完成的状态。这样的业务过程，就称之为“事务”。

事务四大特性：ACID

1.原子性，Atomicity，事务中的多个操作要么都成功，要么都失败，表现的向一个操作一样

2.一致性，Consistency,操作前后数据一致

转账前后，总和不变的

3.隔离性，Isolation,一个事务中看不到另外一个事务过程中的数据

4.持久性，Durability，数据一旦提交，就会持久保存，不会因为断电等导致数据丢失

在用JAVA处理业务时，需要通过事务来控制，流程如下：

try {

与数据库建立连接

修改为手动提交模式

DML操作1;

DML操作2;

DML操作3;

commit;//都成功则提交

} catch(异常){

rollback;//但凡有一个DML有问题，则回滚

} finally{

与数据库断开连接

}

# DDL语句

### 表的创建 – CREATE TABLE

见”DDL表操作”。

补充：通过子查询来创建表。

语法：

CREATE TABLE 表名

AS 子查询;

|  |
| --- |
| CREATE TABLE s\_emp41  AS SELECT id,first\_name,salary,dept\_id  FROM s\_emp  WHERE dept\_id=41; |

注意：通过子查询来创建表，同时包含数据，非空约束会拷贝过来，其他约束丢失。

### 表的修改

**1).增加列**

ALTER TABLE 表名

ADD (

列名 类型 [默认值] [NOT NULL],

...

列名 类型 [默认值] [NOT NULL],

);

注意：可以增加默认值，非空约束，其他约束需要单独加。

**2).修改列**

ALTER TABLE 表名

MODIFY (

列名 类型 [默认值] [NOT NULL],

...

列名 类型 [默认值] [NOT NULL],

);

注意：可以增加默认值，非空约束，其他约束需要单独加。

修改后只要不和原来已存在的值冲突即可。

**3).删除列**

ALTER TABLE 表名

DROP COLUMN 列名;

**4.增加约束,掌握**

ALTER TABLE 表名

ADD CONSTRAINT 约束名 约束类型(修饰的列);

注意：不能增加非空约束。

外键约束：

多个表可能是有关系的，通过外键约束。

被引用的表要先创建，才能创建引用的表，这样对表创建顺序有要求了。

在创建表时，可以先不增加外键约束，对表创建顺序没有要求。

表创建结束后，再添加外键约束。

**5.删除约束**

ALTER TABLE 表名

DROP CONSTRAINT 约束名 [cascade];

cascade:级联删除相关联的约束

**6.使约束失效**

ALTER TABLE 表名

DISABLE CONSTRAINT 约束名 [cascade];

cascade:级联失效和这个约束关联的约束

**7.使约束生效**

ALTER TABLE 表名

ENABLE CONSTRAINT 约束名

### 删除表

DROP TABLE 表名 [cascade constraints];

删除表

cascade constraints：把关联的约束都删除

### 对象重命名

RENAME old\_name TO new\_name

但需要注意的是这个命令是SQL\*Plus的命令, 如果将它放在你自已的程序(如:Java)中, 则是会报错的. 这时应当用标准的重命名语法:

ALTER TABLE old\_name RENAME TO new\_name

### 清空表

TRUNCATE TABLE 表名;

DELETE FROM 表名;

DROP TABLE 表名;

TRUNCATE命令和DELETE的区别：

TRUNCATE : DDL，效率更快，表空间释放，不会恢复。只能用在表。

DELETE: DML，效率不快，表空间没释放，COMMIT才释放，可恢复。除了表，还可以用于视图等。

# 数据字典

## 什么是数据字典？

Oracle数据包括用户表和数据字典两个部分。

数据字典：为了维护系统正常使用，有很多系统创建并管理的表。

数据字典由Oracle数据库创建，更新和维护，包含数据库的相关信息：

1. 数据库的名字。
2. 授权给用户的权限信息。
3. 数据库对象的信息。
4. 表的约束信息。

## 数据字典的分类

根据权限分：

* USER：用户所创建对象对应的数据字典表。

例如：user\_objects, user\_tables, user\_users等。

* ALL：所有用户所能访问对象（包括用户创建的对象）对应的数据字典表。例如： all\_objects, all\_tables, all\_users等。
* DBA：所有对象对应的数据字典表。

例如：dba\_objects, dba\_tables, dba\_users等。

* V$：描述系统性能相关的数据字典表。

例如：v$session, v$parameter等。

* DICTIONARY：一个特殊的数据字典表，用来描述数据字典表相关信息的数据字典表。
* TABLE\_PRIVILEGES：数据表权限。

根据使用人员范围分：

* DBA人员

v$version

dba\_\*

* 开发人员

user\_\*

all\_\*

## 数据字典中常用表的定义

我们需要掌握：

user开头的，当前用户的各种信息

all开头的，当前用户有权限操作的所有用户的信息

user\_users:当前用户相关的信息。

select username from user\_users;//查看当前用户名

效果和show user类似。

user\_tables:当前用户相关的表信息。

查看当前用户下所有的表。

SELECT table\_name FROM user\_tables;

user\_constraints:当前用户相关的约束信息。

constriant\_name

table\_name

查看s\_emp表下面所有的约束：

SELECT constraint\_name

FROM user\_constraints

WHERE table\_name='S\_EMP';

user\_cons\_columns:当前用户相关的约束信息（列）。

constriant\_name

table\_name

column\_name

SELECT constraint\_name,table\_name,column\_name

FROM user\_cons\_columns

WHERE table\_name='S\_EMP';

user\_sequences：当前用户序列。

user\_views：当前用户视图。

user\_indexes：当前用户索引。

# 常见数据库对象

## 序列SEQUENCE

### 什么是序列SEQUENCE？

SEQUENCE（序列）是Oracle数据库中一种特殊的对象，能够产生连续的整数值，可以为数据库中多个对象共同使用，主要用于作为主键值。

当数据库表中添加记录时，通过从序列对象取一个值作为主键列值。

通常，会给每一张表的主键列定义一个对应的序列对象。

### 创建SEQUENCE的语法

CREATE　SEQUENCE <序列名> -- 给序列取名，建议表名-主键名-seq。

[INREMENT BY n] -- 每次增加n，步长，默认是1。

[START WITH n] -- 序列的起始值，默认是1。

[{MAXVALUE n | NOMAXVALUE}] -- 序列的最大值，默认是没有最大值。

[{MINVALUE n | NOMINVALUE}] – 序列的最小值，默认是没有最小值。

[{CYCLE | NOCYCLE}] -- 默认不循环

[{CACHE n | NOCACHE}] -- 默认缓存20个值

[{ORDER | NOORDER}] -- 是否要保证并发取值一定从小到大，通常NOORDER

例1：给学生表t\_student创建一个序列。

|  |
| --- |
| CREATE SEQUENCE t\_student\_id\_seq  START WITH 10  INCREMENT BY 1  CACHE 10; |

### 使用SEQUENCE

创建好SEQUENCE后，通过NEXTVAL或CURRVAL进行调用:

序列名.NEXTVAL : 取序列的下一个值，每次获得不同的序列值。

序列名.CURRVAL : 取序列刚刚取过的值，每次指向当前的序列值。

例2：删除原来创建的成绩表，科目表，学员表。重新创建学员表，并创建学员表的主键序列。利用主键序列，在学员表中增加学生。

|  |
| --- |
| INSERT INTO t\_student (id,name)  VALUES(t\_student\_id\_seq.NEXTVAL,'tom'); |

仅仅只是使用序列NEXTVAL:

SELECT t\_student\_id\_seq.NEXTVAL FROM dual;

### 修改SEQUENCE

修改SEQUENCE的语法：

ALTER SEQUENCE <序列名>

[INREMENT BY n]

[{MAXVALUE n | NOMAXVALUE}]

[{MINVALUE n | NOMINVALUE}]

[{CYCLE | NOCYCLE}]

[{CACHE n | NOCACHE}]

注意：不能修改序列的起始值。

### 删除序列

语法：

DROP　SEQUENCE <序列名>

## 视图和索引

### 什么是视图VIEW？

Oracle视图可以理解为数据库中一张虚拟的表，他是通过一张或者多张基表进行关联查询后组成一个虚拟的逻辑表。查询视图，本质上是对表进行关联查询。

视图的本身是不包含任何数据，只是一个查询结果，当基表的数据发生变化时，视图里面的数据也会跟着发生变化。我们经常在实际开发过程中遇到的视图可以大概分为三种：单表视图、多表关联视图、视图中含有子视图。

### 视图的作用和优势

1)、使数据简单化：可以将复杂的查询创建成视图，提供给他人使用，他人就不需要去理解其中复杂性的业务关系或逻辑关系。这样对视图的使用人员来说，就简化了数据的，屏蔽了数据的复杂性。

2)、表结构设计的补充：系统刚刚开始设计时，大部分程序是直接访问表结构的数据的，但是随着业务的变化、系统的更新等，造成了某些表结构的不适用，这时候去修改表结构对系统的影响太大，开发成本较高，这个时候可以创建视图来对表结构的设计进行补充，降低开发成本。程序可以直接通过查询视图得到想要的数据。

3)、增加安全性：视图可以把表中指定的字段展示给用户，而不必把表中所有字段一起展示给用户。在实际开发中，视图经常作为数据的提供方式，设置为只读权限提供给第三方人员进行查询使用。

### 创建视图

语法：

CREATE [OR REPLACE] VIEW <视图名> --如果视图存在，替换掉旧视图。

AS <子查询>

[WITH READ ONLY] -- 只读

[WITH CHECK POINT] -- 对视图修改时，必须符合视图的where条件

创建视图，需要有权限。Connect和resource并不包含创建视图的权限。需要通过GRANT授予用户创建视图的权限：

GRANT CREATE VIEW TO oaec;

对于简单视图，数据来源于一张表，不包含函数以及分组，可以使用DML操作。

对于复杂视图，数据来源于多张表，包含了函数或分组信息，不可以进行DML操作。

例1：为部门编号41的员工创建一个视图，通过视图可以查看员工编号(id)，名字(first\_name)，薪资(salary)，部门编号(dept\_id)，职称(title)。

|  |
| --- |
| CREATE OR REPLACE VIEW vw\_emp\_dept41  AS SELECT id,first\_name,salary,dept\_id,title  FROM s\_emp  WHERE dept\_id=41  WITH READ ONLY; |

例2：创建一个视图，显示部门名，部门最低薪资，部门最高薪资和部门的平均薪资。

|  |
| --- |
| CREATE VIEW vw\_dept\_sum  (name, minsal, maxsal, avgsal)  AS SELECT d.name, MIN(e.salary),MAX(e.salary), AVG(e.salary)  FROM s\_emp e, s\_dept d  WHERE e.dept\_id = d.id  GROUP BY d.name; |

### 删除视图

语法：

DROP VIEW <视图名>

### 什么是索引INDEX？

索引是ORACLE数据库中的一种对象，数据库服务器用其加快数据检索的速度。

索引会复制加了索引的列值，并将其按照某种算法进行分组排序。

在通过索引列进行检索时，数据库会自动使用索引，进行快速检索，然后通过ROWID快速定位到真实的数据内容。

**自动创建索引：**在创建主键约束，唯一键约束，以及使主键约束、唯一键约束生效时会自动创建索引。

**手动创建索引：**用户可以在列上创建非唯一性的索引。

**注意：**

1. 索引是建立在表上的可选对象；索引的关键在于通过一组排序后的索引键来取代默认的全表扫描检索方式，从而提高检索效率。

2）索引在逻辑上和物理上都与相关的表和数据无关，当创建或者删除一个索引时，不会影响基本的表；

3）索引一旦建立，在表上进行DML操作时（例如在执行插入、修改或者删除相关操作时），oracle会自动管理索引，索引删除，不会对表产生影响。

4）索引对用户是透明的，无论表上是否有索引，sql语句的用法不变。

5）oracle创建主键时会自动在该列上创建索引。

6）创建了索引并不一定就会使用，oracle自动统计表的信息后，决定是否使用索引，表中数据很少时使用全表扫描速度已经很快，没有必要使用索引。

### 索引的分类

索引分类：

* 唯一性索引
* 非唯一性索引
* 单列索引
* 组合索引

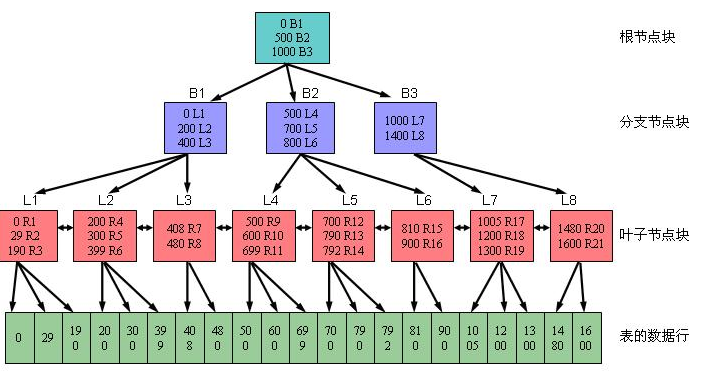
### ROWID

ORACLE数据库的表中的每一行数据都有一个唯一的标识符，或者称为ROWID，在ORACLE内部通常就是使用它来访问数据的。ROWID需要 10个字节的存储空间，并用18个字符来显示。该值表明了该行在ORACLE数据库中的物理具体位置。可以在一个查询中使用rowid来表明查询结果中包含该值。

SELECT ROWID, id FROM s\_emp;

### ORACLE索引原理

在默认情况下，ORACLE索引采用的是B\*TREE结构。



对于分支节点块（包括根节点块）来说，其所包含的索引条目都是按照顺序排列的（缺省是升序排列，也可以在创建索引时指定为降序排列）。每 个索引条目（也可以叫做每条记录）都具有两个字段。第一个字段表示当前该分支节点块下面所链接的索引块中所包含的最小键值；第二个字段为四个字节，表示所 链接的索引块的地址，该地址指向下面一个索引块。在一个分支节点块中所能容纳的记录行数由数据块大小以及索引键值的长度决定。比如从上图一可以看到，对于 根节点块来说，包含三条记录，分别为（0 B1）、（500 B2）、（1000 B3），它们指向三个分支节点块。其中的0、500和1000分别表示这三个分支节点块所链接的键值的最小值。而B1、B2和B3则表示所指向的三个分支节点块的地址。

  对于叶子节点块来说，其所包含的索引条目与分支节点一样，都是按照顺序排列的（缺省是升序排列，也可以在创建索引时指定为降序排列）。每个索引条目（也可以叫做每条记录）也具有两个字段。第一个字段表示索引的键值，对于单列索引来说是一个值；而对于多列索引来说则是多个值组合在一起的。第二个字段表示键值所对应的记录行的ROWID，该ROWID是记录行在表里的物理地址。如果索引是创建在非分区表上或者索引是分区表上的本地索引的话，则该ROWID占用6个字节；如果索引是创建在分区表上的全局索引的话，则该ROWID占用10个字节。

### 创建索引

语法：

CREATE INDEX <索引名>

ON <表名>(<列名>[,<列名>,<列名>…,<列名>])

例1：在员工表中创建last\_name的索引。

|  |
| --- |
| CREATE INDEX idx\_s\_emp\_last\_name  ON s\_emp(last\_name); |

确认创建的索引：

|  |
| --- |
| SELECT ic.index\_name, ic.column\_name, ic.column\_position col\_pos, ix.uniqueness  FROM user\_indexes ix, user\_ind\_columns ic  WHERE ic.index\_name = ix.index\_name  AND ic.table\_name = 'S\_EMP'; |

### 删除索引

语法：

DROP INDEX <索引名>;

删除刚刚创建的索引：

|  |
| --- |
| DROP INDEX idx\_s\_emp\_last\_name; |

# ORACLE表空间

## 理解表空间

Oracle数据库开创性地提出了表空间的设计理念，这为Oracle数据库的高性能做出了不可磨灭的贡献。可以这么说，Oracle中很多优化都是基于表空间的设计理念而实现的。

ORACLE数据库被划分成称作为表空间的逻辑区域——形成ORACLE数据库的[逻辑结构](https://baike.baidu.com/item/%E9%80%BB%E8%BE%91%E7%BB%93%E6%9E%84" \t "_blank)。一个ORACLE数据库能够有一个或多个表空间,而一个表空间则对应着一个或多个物理的[数据库文件](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E6%96%87%E4%BB%B6" \t "_blank)。表空间是ORACLE[数据库恢复](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E6%81%A2%E5%A4%8D)的最小单位,容纳着许多数据库实体,如表、视图、索引、聚簇、回退段和临时段等。

每个ORACLE数据库均有[SYSTEM](https://baike.baidu.com/item/SYSTEM/15078601" \t "_blank)表空间,这是数据库创建时自动创建的。SYSTEM表空间必须总要保持联机,因为其包含着数据库运行所要求的基本信息(关于整个数据库的[数据字典](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%AD%97%E5%85%B8" \t "_blank)、联机求助机制、所有回退段、临时段和自举段、所有的用户数据库实体、其它ORACLE软件产品要求的表)。

一个小型应用的ORACLE数据库通常仅包括SYSTEM表空间,然而一个稍大型应用的ORACLE数据库采用多个表空间会对数据库的使用带来更大的方便。

例如：便于理解，把oracle数据库看作一个实在房间，表空间可以看作这个房间的空间，是可以自由分配，在这空间里面可以堆放多个箱子（箱子可以看作数据库文件），箱子里面再装物件（物件看作表）。用户指定表空间也就是你希望把属于这个用户的表放在那个房间（表空间）里面。  
　　表空间是一个虚拟的概念可以无限大，但是需要由数据文件作为载体。

TIPS:

|  |
| --- |
| 查看表空间：  SQL> select \* from **v$tablespace;**  查看表空间名字：  SQL> select name from v$tablespace;  查看表空间有哪些数据文件：  SQL> select file\_name,tablespace\_name from **dba\_data\_files;** |

## 表空间的分类

永久表空间：主要用于永久化存储的数据库对象。例如：表，视图，存储过程...

临时表空间：主要用于存放一些数据库操作当中，中间执行的过程，当执行结束后，那么存放的内容就会释放掉。它不会永久存放。

UNDO表空间：用于保存事务所修改的数据的旧值。也就是说，被修改之前的数据。比如：当我们修改一个表的时候，它会将修改之前的数据进行保存。这样的话，我们就可以对数据进行回滚。也就说对数据执行撤销操作。

## 查看用户表空间

dba\_tablespaces、user\_tablespaces -- 数据字典

dba\_tablespaces:针对于系统用户查看的数据字典。

user\_tablespaces:针对于普通用户查看的数据字典。

TIPS:

数据库管理员（Database Administrator，简称DBA）

实例：

system用户登陆：

|  |
| --- |
| desc dba\_tablespaces;  select tablespace\_name from dba\_tablespaces; |

普通用户Jason登陆：

|  |
| --- |
| desc user\_tablespaces;  select tablespace\_name from user\_tablespaces; |

## 设置用户默认或临时表空间

查看当前的默认表空间：

|  |
| --- |
| SELECT PROPERTY\_VALUE  FROM **database\_properties**  WHERE PROPERTY\_NAME ='DEFAULT\_PERMANENT\_TABLESPACE'; |

创建用户时，不指定默认表空间，则该用户使用系统指定的默认表空间。

|  |
| --- |
| CREATE USER Jason identified by abc123;  GRANT connect, resource to Jason;  Conn Jason/abc123;  查看默认空间，用户空间。命令略 |

系统默认给用户的默认表空间：system

TIPS

|  |
| --- |
| 可以使用如下语句修改系统默认安排给用户的默认表空间：  ALTER DATABASE DEFAULT TABLESPACE users;  \*这样，如果创建用户时，不再指定默认表空间，则用户的默认表空间为 users , 不再是system.  \*如果我们在创建用户时指定了默认表空间，那么在修改默认表空间后，之前用户的默认表空间也会发生改变。  \*如果我们在创建用户指定用户的表空间是其他的表空间，那么我们修改DB的默认表空间不会影响用户的表空间。 |

创建用户时指定用户的默认表空间：

CREATE USER Jason IDENTIFIED BY abc123 **DEFAULT TABLESPACE users;**

TIPS 表空间有关查询的语句：

|  |
| --- |
| 1、查看所有的表空间：SELECT \* FROM DBA\_TABLESPACES;  2、查看某个用户的默认表空间：SELECT DEFAULT\_TABLESPACE,USERNAME FROM DBA\_USERS WHERE USERNAME='username';  3、查看表空间剩余容量：SELECT TABLESPACE\_NAME,SUM(BYTES) FROM DBA\_FREE\_SPACE GROUP BY TABLESPACE\_NAME;  4、查看表空间数据文件的信息：SELECT \* FROM DBA\_DATA\_FILES;  5、创建表空间：CREATE TABLESPACE ODI  DATAFILE 'D:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\ODI.DBF ' SIZE 50M AUTOEXTEND ON NEXT 10M PERMANENT EXTENT MANAGEMENT LOCAL;  完整的建表空间语句：CREATE TABLESPACE tablespace\_name DATAFILE 'filename' SIZE size [AUTOEXTEND [ON NEXT size | OFF]] [MAXSIZE size][PERMANENT  | TEMPORARY][EXTENT MANAGEMENT DICTIONARY | LOCAL];  6、重命名表空间：ALTER TABLESPACE oldname RENAME TO newname; |

## 创建表空间

创建表空间oaecspace：

|  |
| --- |
| CREATE TABLESPACE oaecspace  DATAFILE 'F:\oracledata\oaec\_dbspace.dbf'  SIZE 100M  AUTOEXTEND ON NEXT 2M MAXSIZE 256M  EXTENT MANAGEMENT LOCAL; |

创建临时表空间：

|  |
| --- |
| CREATE TEMPORARY TABLESPACE oaec\_temp  TEMPFILE 'F:\oracledata\oaec\_temp.dbf'  SIZE 16M  AUTOEXTEND ON NEXT 2M MAXSIZE 128M  EXTENT MANAGEMENT LOCAL; |

创建用户时，指定表空间和临时表空间：

CREATE USER 用户名 IDENTIFIED BY 口令

DEFAULT TABLESPACE 表空间名

TEMPORARY TABLESPACE 临时表空间名;

|  |
| --- |
| CREATE USER test IDENTIFIED BY test  DEFAULT TABLESPACE oaecspace  TEMPORARY TABLESPACE oaec\_temp; |

## 删除表空间

语法：

DROP TABLESPACE <表空间>

INCLUDING CONTENTS AND DATAFILES;