

第08章_聚合函数

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

我们上一章讲到了 SQL 单行函数。实际上 SQL 函数还有一类，叫做聚合（或聚集、分组）函数，它是**对一组数据进行汇总的函数，输入的是一组数据的集合，输出的是单个值。**

1. 聚合函数介绍

- 什么是聚合函数

聚合函数作用于一组数据，并对一组数据返回一个值。

EMPLOYEES

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

...

20 rows selected.

表 EMPLOYEES
中的工资最大值

MAX(SALARY)
24000

- 聚合函数类型

- AVG()
- SUM()
- MAX()
- MIN()
- COUNT()

- 聚合函数语法

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

- 聚合函数不能嵌套调用。比如不能出现类似“AVG(SUM(字段名称))”形式的调用。

1.1 AVG和SUM函数

可以对数值型数据使用AVG 和 SUM 函数。

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

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

1.2 MIN和MAX函数

可以对任意数据类型的数据使用 MIN 和 MAX 函数。

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

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

1.3 COUNT函数

- COUNT(*)返回表中记录总数，适用于任意数据类型。

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

COUNT(*)
5

- COUNT(expr) 返回expr不为空的记录总数。

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

COUNT(COMMISSION_PCT)
3

- 问题：用count(*), count(1), count(列名)谁好呢？

其实，对于MyISAM引擎的表是没有区别的。这种引擎内部有一计数器在维护着行数。

InnoDB引擎的表用count(*),count(1)直接读行数，复杂度是O(n)，因为innodb真的要去数一遍。但好于具体的count(列名)。

- 问题：能不能使用count(列名)替换count(*)？

不要使用 count(列名)来替代 count(*)，count(*) 是 SQL92 定义的标准统计行数的语法，跟数据库无关，跟 NULL 和非 NULL 无关。

说明：count(*)会统计值为 NULL 的行，而 count(列名)不会统计此列为 NULL 值的行。

COUNT(*)和COUNT(1)在InnoDB中都不需要访问实际的列数据，而COUNT(列名)需要检查每一行该字段的值是否为NULL

2. GROUP BY

2.1 基本使用

EMPLOYEES				
DEPARTMENT_ID		SALARY		
10		4400	4400	
20		13000		
20		6000	9500	
50		5800		
50		3500		
50		3100	3500	
50		2500		
50		2600		
60		9000		
60		6000	6400	
60		4200		
80		10500		
80		8600		
80		11000	10033	
90		24000		
90		17000		
...				
20 rows selected.				

求出
EMPLOYEES
表中各
部门的
平均工资

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

可以使用GROUP BY子句将表中的数据分成若干组

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

明确：WHERE一定放在FROM后面

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

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

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

8 rows selected.

包含在 GROUP BY 子句中的列不必包含在SELECT 列表中

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

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

2.2 使用多个列分组

EMPLOYEES

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

20 rows selected.

使用多个列
进行分组

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

13 rows selected.

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

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

13 rows selected.

2.3 GROUP BY中使用WITH ROLLUP

使用 **WITH ROLLUP** 关键字之后，在所有查询出的分组记录之后增加一条记录，~~该记录计算查询出的所有记录的总和，即统计记录数量。~~

```
SELECT department_id,AVG(salary)
FROM employees
WHERE department_id > 80
GROUP BY department_id WITH ROLLUP;
```

该记录计算不分组时，所有满足条件的记录的AVG/SUM/COUNT等

注意：

~~当使用ROLLUP时，不能同时使用ORDER BY子句进行结果排序，即ROLLUP和ORDER BY是互相排斥的。~~

ROLLUP和ORDER BY可以一起用，只不过是把整个表的结果与分组结果混合在一起排序

3. HAVING

3.1 基本使用

EMPLOYEES

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

20 rows selected.

部门最高工资
比 10000 高的
部门

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

过滤分组：HAVING子句

1. 行已经被分组。
2. 使用了聚合函数。
3. 满足HAVING子句中条件的分组将被显示。
4. **HAVING 不能单独使用，必须要跟 GROUP BY 一起使用。**

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

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

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

- **非法使用聚合函数：**不能在 WHERE 子句中使用聚合函数。如下：

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

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

3.2 WHERE和HAVING的对比

区别1：WHERE 可以直接使用表中的字段作为筛选条件，但不能使用分组中的计算函数作为筛选条件；HAVING 必须要与 GROUP BY 配合使用，可以把分组计算的函数和分组字段作为筛选条件。

这决定了，在需要对数据进行分组统计的时候，HAVING 可以完成 WHERE 不能完成的任务。这是因为，在查询语法结构中，WHERE 在 GROUP BY 之前，所以无法对分组结果进行筛选。HAVING 在 GROUP BY 之后，可以使用分组字段和分组中的计算函数，对分组的结果集进行筛选，这个功能是 WHERE 无法完成的。另外，WHERE 排除的记录不再包括在分组中。

区别2：如果需要通过连接从关联表中获取需要的数据，WHERE 是先筛选后连接，而 HAVING 是先连接后筛选。这一点，就决定了在关联查询中，WHERE 比 HAVING 更高效。因为 WHERE 可以先筛选，用一个筛选后的较小数据集和关联表进行连接，这样占用的资源比较少，执行效率也比较高。HAVING 则需要先把结果集准备好，也就是用未被筛选的数据集进行关联，然后对这个大的数据集进行筛选，这样占用的资源就比较多，执行效率也较低。

小结如下：

	优点	缺点
WHERE	先筛选数据再关联，执行效率高	不能使用分组中的计算函数进行筛选
HAVING	可以使用分组中的计算函数	在最后的結果集中进行筛选，执行效率较低

开发中的选择：

WHERE 和 HAVING 也不是互相排斥的，我们可以在一个查询里面同时使用 WHERE 和 HAVING。包含分组统计函数的条件用 HAVING，普通条件用 WHERE。这样，我们就既利用了 WHERE 条件的高效快速，又发挥了 HAVING 可以使用包含分组统计函数的查询条件的优点。当数据量特别大的时候，运行效率会有很大的差别。

4. SELECT的执行过程

4.1 查询的结构

```
#方式1：
SELECT ..., ..., ...
FROM ..., ..., ...
WHERE 多表的连接条件
AND 不包含组函数的过滤条件
GROUP BY ..., ...
HAVING 包含组函数的过滤条件
ORDER BY ... ASC/DESC
LIMIT ..., ...

#方式2：
SELECT ..., ..., ...
FROM ... JOIN ...
ON 多表的连接条件
JOIN ...
ON ...
WHERE 不包含组函数的过滤条件
AND/OR 不包含组函数的过滤条件
GROUP BY ..., ...
HAVING 包含组函数的过滤条件
ORDER BY ... ASC/DESC
LIMIT ..., ...

#其中：
#（1）from：从哪些表中筛选
#（2）on：关联多表查询时，去除笛卡尔积
#（3）where：从表中筛选的条件
#（4）group by：分组依据
#（5）having：在统计结果中再次筛选
#（6）order by：排序
#（7）limit：分页
```

4.2 SELECT执行顺序

你需要记住 SELECT 查询时的两个顺序：

1. 关键字的顺序是不能颠倒的：

```
SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ... ORDER BY ... LIMIT...
```

2. SELECT 语句的执行顺序（在 MySQL 和 Oracle 中，SELECT 执行顺序基本相同）：

```
FROM -> WHERE -> GROUP BY -> HAVING -> SELECT 的字段 -> DISTINCT -> ORDER BY -> LIMIT
```

```
1 FROM <left_table>
2 ON <join_condition>
3 <join_type> JOIN <right_table>
4 WHERE <where_condition>
5 GROUP BY <group_by_list>
6 HAVING <having_condition>
7 SELECT
8 DISTINCT <select_list>
9 ORDER BY <order_by_condition>
10 LIMIT <limit_number>
```

比如你写了一个 SQL 语句，那么它的关键字顺序和执行顺序是下面这样的：

```
SELECT DISTINCT player_id, player_name, count(*) as num # 顺序 5
FROM player JOIN team ON player.team_id = team.team_id # 顺序 1
WHERE height > 1.80 # 顺序 2
GROUP BY player.team_id # 顺序 3
HAVING num > 2 # 顺序 4
ORDER BY num DESC # 顺序 6
LIMIT 2 # 顺序 7
```

在 SELECT 语句执行这些步骤的时候，每个步骤都会产生一个虚拟表，然后将这个虚拟表传入下一个步骤中作为输入。需要注意的是，这些步骤隐含在 SQL 的执行过程中，对于我们来说是不可见的。

4.3 SQL 的执行原理

SELECT 是先执行 FROM 这一步的。在这个阶段，如果是多张表联查，还会经历下面的几个步骤：

1. 首先通过 CROSS JOIN 求笛卡尔积，相当于得到虚拟表 vt (virtual table) 1-1；
2. 通过 ON 进行筛选，在虚拟表 vt1-1 的基础上进行筛选，得到虚拟表 vt1-2；
3. 添加外部行。如果我们使用的是左连接、右链接或者全连接，就会涉及到外部行，也就是在虚拟表 vt1-2 的基础上增加外部行，得到虚拟表 vt1-3。

当然如果我们操作的是两张以上的表，还会重复上面的步骤，直到所有表都被处理完为止。这个过程得到的是我们的原始数据。

当我们拿到了查询数据表的原始数据，也就是最终的虚拟表 vt1，就可以在此基础上再进行 WHERE 阶段。在这个阶段中，会根据 vt1 表的结果进行筛选过滤，得到虚拟表 vt2。

然后进入第三步和第四步，也就是 GROUP 和 HAVING 阶段。在这个阶段中，实际上是在虚拟表 vt2 的基础上进行分组和分组过滤，得到中间的虚拟表 vt3 和 vt4。

当我们完成了条件筛选部分之后，就可以筛选表中提取的字段，也就是进入到 SELECT 和 DISTINCT 阶段。

首先在 SELECT 阶段会提取想要的字段，然后在 DISTINCT 阶段过滤掉重复的行，分别得到中间的虚拟表 vt5-1 和 vt5-2。

当我们提取了想要的字段数据之后，就可以按照指定的字段进行排序，也就是 ORDER BY 阶段，得到虚拟表 vt6。

最后在 vt6 的基础上，取出指定行的记录，也就是 LIMIT 阶段，得到最终的结果，对应的是虚拟表 vt7。

当然我们在写 SELECT 语句的时候，不一定存在所有的关键字，相应的阶段就会省略。

同时因为 SQL 是一门类似英语的结构化查询语言，所以我们在写 SELECT 语句的时候，还要注意相应的关键字顺序，**所谓底层运行的原理，就是我们刚才讲到的执行顺序。**