

04 学会聚合与分组聚合是很有必要的

更新时间：2020-08-10 14:42:59



“受苦的人，没有悲观的权利。——尼采”

在我们日常的工作中，“函数”这个概念肯定不会陌生。例如，我们使用 **Java** 语言时，可以使用 **JDK** 自带的函数、也可以使用依赖的其他 **jar** 包中的函数、还可以自定义函数等等。为了方便我们的工作，**MySQL** 同样提供了不同种类的函数。这一节里，我们来探讨下 **MySQL** 的“聚合函数”，以及怎样使用 **GROUP BY** 语句实现分组聚合。

1 理解聚合函数

想要把聚合函数使用好，首先需要知道什么是聚合函数？常用的聚合函数又有哪些？它们的使用方法又是怎样的？下面，我将会介绍聚合函数的概念、常用的聚合函数及语法，再辅以实践案例详细的说明聚合函数。

1.1 聚合函数的概念

在数据库中，函数可以分为两种：单行函数和多行函数。单行函数即函数会针对每一行返回一个结果，而多行函数则是作用于多行（也可以作用于单行）并返回一个结果。聚合函数则属于多行函数，表中的多行记录会参与计算，并返回一个数值，且它通常用于分组的相关统计。

1.2 常用的聚合函数

MySQL 的官方文档中给出了非常多的聚合函数，但其中的大多数在我们的日常工作中都是用不到的。常用的聚合函数有五个：**AVG**、**COUNT**、**MIN**、**MAX**、**SUM**。下面，我们来看一看它们的语法及含义。

语法	功能	备注
AVG ([DISTINCT] expr)	返回 expr 的平均值	DISTINCT 选项用于去除字段值重复的行记录
COUNT(expr)	统计表中的行数	
MIN ([DISTINCT] expr)	返回 expr 的最小值	
MAX ([DISTINCT] expr)	返回 expr 的最大值	
SUM ([DISTINCT] expr)	返回 expr 的合计值	

可以看出，这些聚合函数的定义虽然简单，但是功能强大：选好聚合类型并给定参数（**expr**）就可以得到聚合结果。关于这些聚合函数，在使用的过程中需要知道它们的一些共性：

- 每个聚合函数接受一个参数，参数可以是数据表列，也可以是函数表达式
- 默认情况下，聚合函数会忽略列值为 **NULL** 的行，不参与计算
- 聚合函数不允许嵌套，例如：**COUNT(SUM(expr))** 是不合法的
- 一次查询中可以出现多个聚合函数，例如：**SELECT MAX(expr), MIN(expr) FROM ...**

2 聚合函数的实践

想要更好的理解技术，就一定要去深入实践，理论结合实践才是最好的学习方式。为了更方便的讲解聚合函数的使用方法，我们假定数据表 **worker** 中存储了如下的数据。

```
mysql> SELECT id, type, name, salary FROM worker;
+----+-----+-----+-----+
| id | type | name  | salary |
+----+-----+-----+-----+
| 1  | A    | tom   | 1800   |
| 2  | B    | jack  | 2100   |
| 3  | C    | pony  | NULL   |
| 4  | B    | tony  | 3600   |
| 5  | B    | marry | 1900   |
| 6  | C    | tack  | 1200   |
| 7  | A    | tick  | NULL   |
| 8  | B    | clock | 2000   |
| 9  | C    | noah  | 1500   |
| 10 | C    | jarvis | 1800   |
+----+-----+-----+-----+
```

需要注意的是，**id** 为3和7的记录中，**salary** 列值都是 **NULL**。根据之前讲述的聚合函数的特性，当 **expr** 传入这一列时，这两条记录则不会参与计算。

2.1 使用 **AVG** 计算平均值

AVG 只适用于数值类型的列，因为对于像日期、字符串等类型求平均本身就是没有意义的。例如，我们可以使用 **AVG** 计算所有 **worker salary** 的均值。

```
mysql> SELECT AVG(salary) FROM worker;
+-----+
| AVG(salary) |
+-----+
| 1987.5000 |
+-----+
```

所有的聚合函数，如果是以列名作为 **expr**（例如这里的 **salary**），MySQL 会在计算之前把列值是 **NULL** 的记录排除掉。这里需要理解，不能认为“**NULL** 是0”，实际是列值为 **NULL** 的行不会计入分母。

另外，使用聚合函数的同时也并不妨碍条件查询，我们同样可以使用 **WHERE** 条件先对行记录做筛选，再去计算平均值。例如：

```
mysql> SELECT AVG(salary) FROM worker WHERE id < 3;
+-----+
| AVG(salary) |
+-----+
| 1950.0000 |
+-----+
```

2.2 使用 COUNT 计算表中的行数

COUNT 函数相对于其他聚合函数来说，是比较特殊的，它的使用方法比较多，通常可以看到的用法包括：**COUNT(n)**、**COUNT(*)**、**COUNT(expr)**、**COUNT(DISTINCT expr)**。所以，接下来我们需要探究下这几种方式的含义、特性与适用场景。

COUNT(n) 中的 **n** 可以是任何整数或小数，它与 **COUNT(*)** 的查询结果是一样的，例如：

```
mysql> SELECT COUNT(0), COUNT(1), COUNT(9.9), COUNT(*) FROM worker;
+-----+-----+-----+-----+
| COUNT(0) | COUNT(1) | COUNT(9.9) | COUNT(*) |
+-----+-----+-----+-----+
| 10 | 10 | 10 | 10 |
+-----+-----+-----+-----+
```

另外，从输出中还可以得出结论，**COUNT(n)** 和 **COUNT(*)** 统计的总行数是包含 **NULL** 值的。这两种统计方式从本质上来说是一样的，而且并不存在哪一种效率更高的说法。这两种统计方法都不会使用全表扫描，而是使用了 **PRIMARY** 索引优化查询，性能是非常高的。所以，想要查询表中的记录行数，使用它们之中的任何一个都是可以的。

COUNT(expr) 和 **COUNT(DISTINCT expr)** 由于需要传入列作为参数，所以，它们统计的是非 **NULL** 的行数。如果加上了 **DISTINCT**，则是统计列值不相同且非 **NULL** 的行数。验证如下：

```
mysql> SELECT COUNT(salary), COUNT(DISTINCT salary) FROM worker;
+-----+-----+
| COUNT(salary) | COUNT(DISTINCT salary) |
+-----+-----+
| 8 | 7 |
+-----+-----+
```

由于 **id** 是3和7的 **salary** 列值是 **NULL**，所以，**COUNT(salary)** 的结果是8。又由于 **id**是1和10的 **salary** 值相同，所以，排除一个，最终 **COUNT(DISTINCT salary)** 的结果是7。

关于 **COUNT** 函数，总结如下：

- **COUNT(n)** 和 **COUNT(*)** 用于统计表中的总行数，不关心列值是否为 **NULL**
- **COUNT(expr)** 用于统计列值非 **NULL** 的行记录数
- **COUNT(DISTINCT expr)** 用于统计列值不同且非 **NULL** 的行记录数

2.3 使用 MIN、MAX 计算最小值、最大值

MIN、MAX 函数适用于任何能够排序的数据，注意，不同于 AVG，它们的适用范围不只是数值类型，日期类型、字符串类型也同样是允许的。由于这两个函数的功能、使用方法相对来说比较简单，这里给出一个例子，不再多做说明。

```
mysql> SELECT MIN(salary), MAX(salary) FROM worker;
+-----+-----+
| MIN(salary) | MAX(salary) |
+-----+-----+
|      1200 |       3600 |
+-----+-----+
```

2.4 使用 SUM 计算合计值

SUM 函数正如同这个单词的字面意思，用于计算列的合计值（总值）。它几乎与 AVG 有一样的性质：只能用于数值类型的列，且会忽略值为 NULL 的列。例如：

```
mysql> SELECT SUM(salary) FROM worker WHERE id < 5;
+-----+
| SUM(salary) |
+-----+
|       7500 |
+-----+
```

另外，大家可能会看到 SUM(1) 这样的语法，它的作用与 COUNT(n) 或 COUNT(*) 是相同的，都是用来统计行记录数。但是，从效率上来说，SUM(1) 是非常慢的，我们应该尽量避免这种用法。

```
mysql> SELECT SUM(1) FROM worker;
+-----+
| SUM(1) |
+-----+
|      10 |
+-----+
```

同时，需要注意，SUM(2) 或者是其他的数字，得到的并不是行记录数。你可以简单的理解为：SUM 操作会遍历整个表，遇到一条记录，就会执行一次加 N 的操作，最终返回累加和，即行记录数的 N 倍。

3 掌握分组聚合

分组的意思就是数据根据某一列或者某几列分类，MySQL 中可以使用 GROUP BY 子句实现这一功能。GROUP BY 结合聚合函数就可以实现将表数据分类再汇总的效果，这在报表型的数据统计任务中是非常常见的需求。

GROUP BY 子句的语法如下：

```
SELECT
  <列名1>,
  <列名2>.....
FROM
  <表名>
WHERE
  .....
GROUP BY
  <列名1>,
  <列名2>.....;
```

GROUP BY 子句中的列称为聚合列或分组列。下面，我将用一些实例说明分组聚合的含义、使用方法、特性与需要注意的地方（同样使用之前的 worker 表作为示例数据）。

3.1 按照 type 分组对数据进行统计

对于 worker 表来说，我们可以按照 type 对数据记录进行分组，分组之后再按照想要统计的类型进行聚合操作。例如：

```
mysql> SELECT type, AVG(salary), COUNT(1), MIN(salary), MAX(salary), SUM(salary) FROM worker GROUP BY type;
```

type	AVG(salary)	COUNT(1)	MIN(salary)	MAX(salary)	SUM(salary)
A	1800.0000	2	1800	1800	1800
B	2400.0000	4	1900	3600	9600
C	1500.0000	4	1200	1800	4500

使用 GROUP BY 对 type 字段值进行分组，结果有三类：A、B、C。分组之后，AVG、COUNT 等聚合函数再按照自身的特性对每一组数据进行聚合统计，最后，打印如上结果。

需要注意的是，出现在 SELECT 子句中的单独列（非聚合列，示例中的即为 type），必须出现在 GROUP BY 子句中作为分组列。但是反过来，分组列是可以不出现在 SELECT 子句中的。

3.2 对分组聚合结果进行排序

分组聚合的结果没有什么特殊之处，当然也是可以指定排序的。指定排序的列可以是分组列，也可以不是分组列。

例如，我们可以按照 SUM(salary) 实现排序：

```
mysql> SELECT type, SUM(salary) as sum_s FROM worker GROUP BY type ORDER BY sum_s desc;
```

type	sum_s
B	9600
C	4500
A	1800

有一种特殊情况，当排序列与分组列相同时，则可以合并 GROUP BY 和 ORDER BY 子句，即只需要在 GROUP BY 子句的后面添加 DESC 或 ASC。例如：

```
mysql> SELECT type, AVG(salary), COUNT(1), MIN(salary), MAX(salary), SUM(salary) FROM worker GROUP BY type DESC;
```

type	AVG(salary)	COUNT(1)	MIN(salary)	MAX(salary)	SUM(salary)
C	1500.0000	4	1200	1800	4500
B	2400.0000	4	1900	3600	9600
A	1800.0000	2	1800	1800	1800

3.3 对分组结果进行过滤

这个标题其实是有误导性的，大家需要仔细审题。这里过滤的是分组后的聚合结果，而不是数据表中的原始记录。

在 MySQL 中，使用 AVG、COUNT 等聚合函数对表记录进行统计操作后，可以使用 HAVING 子句对结果进行过滤，且 HAVING 子句需要写在 GROUP BY 子句之后。例如，我们按照 type 对 worker 表中的数据分组之后，想要获取 SUM(salary) 大于 4000 的分组，可以这样做：

```
mysql> SELECT type, AVG(salary), COUNT(1), SUM(salary) FROM worker GROUP BY type HAVING SUM(salary) > 4000;
```

type	AVG(salary)	COUNT(1)	SUM(salary)
B	2400.0000	4	9600
C	1500.0000	4	4500

可以看到，HAVING 的使用方法与 WHERE 是相似的，只是它们执行的时机不同。总结下来，它们有以下两个区别：

- WHERE 子句在分组前对记录进行过滤
- HAVING 子句在分组后对记录进行过滤

分组聚合的精髓在于数据分组，可以把每一个分组都认为是单独的数据表记录，最终的聚合结果则是将每一个单独数据表聚合之后 merge 而成的。另外，需要知道，聚合函数可以在 SELECT、HAVING 和 ORDER BY 子句中使用，但是不能在 WHERE 子句中使用。

4 工作中的实例

学以致用用的最佳应用场景肯定是在工作中，这里我将给出一些实例，同时也是我在平时的工作中所遇到的一些需求，并使用聚合与分组聚合解决的案例。首先，我将给出表的创建语句，以此能够知道表结构组成。

```
CREATE TABLE `ad_unit` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '自增主键',
  `user_id` bigint(20) NOT NULL DEFAULT '0' COMMENT '标记当前记录所属用户',
  `unit_name` varchar(48) NOT NULL COMMENT '推广单元名称',
  `unit_status` tinyint(4) NOT NULL DEFAULT '0' COMMENT '推广单元状态: 0-正常, 1-失效',
  `position_type` tinyint(4) NOT NULL DEFAULT '0' COMMENT '广告位类型(1,2,3)',
  `budget` bigint(20) NOT NULL COMMENT '预算(单位: 元)',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='广告-推广单元表';
```

当前表中存储的数据如下：

```
mysql> SELECT * FROM ad_unit;
```

id	user_id	unit_name	unit_status	position_type	budget
1	1001	推广单元-1	0	1	1200
2	1001	推广单元-2	0	1	1500
3	1001	推广单元-3	0	2	1700
4	1001	推广单元-4	1	1	2500
5	1002	推广单元-5	0	1	2000
6	1002	推广单元-6	0	3	1000
7	1003	推广单元-7	0	1	3400
8	1003	推广单元-8	0	2	2100
9	1004	推广单元-9	0	1	1600
10	1004	推广单元-10	0	1	1100
11	1004	推广单元-11	0	2	3500
12	1004	推广单元-12	0	3	1900
13	1004	推广单元-13	0	3	3200

4.1 查询某个/所有用户的最大/小预算

这是个很常见的需求，目的就是看看广告主（对应到表中的 `user_id`）预算的极值。同时，这个需求也是非常简单的：某个用户的话使用 `WHERE` 子句先去筛选用户记录即可；所有用户的话，首先根据 `user_id` 做好分组，再去使用聚合函数即可。SQL 语句实现如下：

```
-- 查询 user_id 是 1004 的最大/小预算, 需要排除无效的记录
mysql> SELECT user_id, MAX(budget), MIN(budget) FROM ad_unit WHERE user_id = 1004 AND unit_status = 0;
+-----+-----+-----+
| user_id | MAX(budget) | MIN(budget) |
+-----+-----+-----+
| 1004    | 3500        | 1100        |
+-----+-----+-----+
```

```
-- 查询所有用户的最大/最小预算, 需要排除无效的记录
mysql> SELECT user_id, MAX(budget), MIN(budget) FROM ad_unit WHERE unit_status = 0 GROUP BY user_id;
+-----+-----+-----+
| user_id | MAX(budget) | MIN(budget) |
+-----+-----+-----+
| 1001    | 1700        | 1200        |
| 1002    | 2000        | 1000        |
| 1003    | 3400        | 2100        |
| 1004    | 3500        | 1100        |
+-----+-----+-----+
```

4.2 查询所有用户的分类广告位类型最大/小预算

对于需求中的所有用户来说，我们并不陌生，只需要使用 `GROUP BY` 子句按照 `user_id` 分组即可。但是注意到，这里除了分组用户之外，还需要对广告位类型进行分组。这其实也很简单，因为 `GROUP BY` 子句的语法是支持对多个列进行分组的。如下：

```
mysql> SELECT user_id, position_type, MAX(budget), MIN(budget) FROM ad_unit WHERE unit_status = 0 GROUP BY user_id, position_type;
+-----+-----+-----+-----+
| user_id | position_type | MAX(budget) | MIN(budget) |
+-----+-----+-----+-----+
| 1001    | 1             | 1500        | 1200        |
| 1001    | 2             | 1700        | 1700        |
| 1002    | 1             | 2000        | 2000        |
| 1002    | 3             | 1000        | 1000        |
| 1003    | 1             | 3400        | 3400        |
| 1003    | 2             | 2100        | 2100        |
| 1004    | 1             | 1600        | 1100        |
| 1004    | 2             | 3500        | 3500        |
| 1004    | 3             | 3200        | 1900        |
+-----+-----+-----+-----+
```

4.3 查询总预算超过 5000 的用户

显然这个需求是要计算 `budget` 的合计值，且在计算聚合（`SUM`）之前需要先对记录按照 `user_id` 进行分组。另外，需求中的 5000（可以是任意数字）指的是聚合之后的值，所以，应该想到这里需要对结果执行 `HAVING` 计算。如下：

```
mysql> SELECT user_id, SUM(budget) FROM ad_unit WHERE unit_status = 0 GROUP BY user_id HAVING SUM(budget) > 5000;
+-----+-----+
| user_id | SUM(budget) |
+-----+-----+
| 1003    | 5500        |
| 1004    | 11300       |
+-----+-----+
```

从以上几个实例中可以看出，需求本身往往都不会很复杂，只要理解了聚合与分组聚合的核心知识点，剩下的就是按步骤拆解需求，选择正确的聚合函数与执行子句就可以了。

5 总结

聚合与分组聚合是 MySQL 基础知识中非常重要的部分，它们大量的出现在报表型应用中，做 OLAP 操作。想要学好聚合函数，先要把每一个聚合函数的概念、适用场景搞清楚，再去理解它们的特性，最后多做实践。而分组聚合是先分组再聚合，本质上说，还是对多行数据做聚合统计操作。所以，更重要的是理解分组的含义。

6 问题

为什么说 SUM(1) 的执行效率要比 COUNT(n) 或 COUNT(*) 低很多呢？

SELECT、FROM、WHERE、HAVING、GROUP BY、ORDER BY 这些子句的正确书写顺序应该是怎样的？

7 参考资料

[MySQL 官方文档](#)

}