处理高级SQL



在第3章 "利用索引 "中，你了解了索引，以及PostgreSQL运行自定义索引代码以加速查询的能力。在本章中，你将学习高级SQL。阅读本书的大多数人都会有一些使用SQL的经验。然而，经验表明，本书所概述的高级功能并不广为人知，因此，在这种情况下介绍这些功能是有意义的，可以帮助人们更快、更有效地实现他们的目标。关于数据库是否只是一个简单的数据存储，或者业务逻辑是否应该在数据库中，已经有很长的讨论。 也许这一章会给我们一些启示，说明现代关系型数据库到底有多大能力。SQL已经不是当年SQL-92时的样子了。多年来，这种语言不断发展，变得越来越强大。

本章是关于现代SQL和它的功能。涵盖并详细介绍了各种不同的、复杂的SQL特性。在本章中，我们将介绍以下内容。

- 介绍分组集
- 使用有序集
- 了解假设的聚合
- 利用窗口功能和分析
- 编写你自己的聚合

在本章结束时，你将了解并能够使用高级SQL。

# 1 介绍分组集

每个SQL的高级用户都应该熟悉GROUP BY和HAVING子句。但他们是否也知道CUBE、ROLLUP和GROUPING SETS？如果没有，这一章是必读的。分组集的基本思想是什么？基本上，这个概念很简单：使用一个分组集，你可以将各种聚合合并到一个查询中。其主要优点是，你只需读取一次数据，同时一次产生许多不同的聚合集。

## 1.1 加载一些样品数据

为了使这一章给你带来愉快的体验，我们将汇编一些样本数据，这些数据取自BP能源报告，可以在http://www.bp.com/en/global/corporate/energy-economics/statistical-review-of-world-energy/downloads.html找到。

以下是将要使用的数据结构。

```
test=# CREATE TABLE t_oil (
 region text,
 country text,
 year int,
 production int,
 consumption int
);
CREATE TABLE
```

测试数据可以直接用curl从我们的网站下载。

```
test=# COPY t_oil FROM PROGRAM '
 curl https://www.cybertec-postgresql.com/secret/oil_ext.txt ';
COPY 644
```

就像我们在前一章所做的那样，我们可以在导入文件之前下载文件。在一些操作系统上，curl默认不存在或者没有安装，所以对很多人来说，在导入文件之前下载文件可能是一个更容易的选择。

我们有一些1965年至2010年的石油生产和消费数据，这些数据来自世界2个地区的14个国家。

```
test=# SELECT region, avg(production)
 FROM t_oil GROUP BY region;
 region | avg
---------------+-----------------------
 Middle East | 1992.6036866359447005
 North America | 4541.3623188405797101
(2 rows)
```

结果正是我们所期望的：两行包含平均产量。

## 1.2 应用分组集

GROUP BY子句会把许多行变成每组的一条行。然而，如果你在现实生活中做报告，用户可能也会对总体的平均数感兴趣。可能需要增加一行。

这就是如何实现的。

```
test=# SELECT region, avg(production)
 FROM t_oil
 GROUP BY ROLLUP (region);
 region | avg
----------------+-----------------------
 Middle East | 1992.6036866359447005
 North America | 4541.3623188405797101
 | 2607.5139860139860140
(3 rows)
```

ROLLUP关键字将注入一个额外的行，该行将包含总体平均数。如果你做报告，很可能需要一个摘要行。与其运行两个查询，PostgreSQL可以通过运行一个查询来提供数据。这里还有第二件事你可能会注意到；不同版本的PostgreSQL可能以不同的顺序返回数据。原因是在PostgreSQL 10.0中，那些分组集的实现方式有了很大的改进。早在9.6版和之前，PostgreSQL不得不做大量的排序。从10.0版本开始，可以使用散列法进行这些操作，这在很多情况下会大大加快速度，如下面的代码块所示。

```
test=# explain SELECT region, avg(production)
 FROM t_oil
 GROUP BY ROLLUP (region);
 QUERY PLAN
-------------------------------------------------------------
 MixedAggregate (cost=0.00..17.31 rows=3 width=44)
 Hash Key: region
 Group Key: ()
 -> Seq Scan on t_oil (cost=0.00..12.44 rows=644 width=16)
(4 rows)
```

如果我们希望对数据进行排序，并确保所有的版本都以完全相同的顺序返回数据，就有必要在查询中添加一个ORDER BY子句。

当然，如果你要按不止一个列进行分组，也可以使用这种操作。

```
test=# SELECT region, country, avg(production)
 FROM t_oil
 WHERE country IN ('USA', 'Canada', 'Iran', 'Oman')
 GROUP BY ROLLUP (region, country);
 region | country | avg
---------------+---------+-----------------------
 Middle East | Iran | 3631.6956521739130435
 Middle East | Oman | 586.4545454545454545
 Middle East | | 2142.9111111111111111
 North America | Canada | 2123.2173913043478261
 North America | USA | 9141.3478260869565217
 North America | | 5632.2826086956521739
 | | 3906.7692307692307692
(7 rows)
```

在前面的例子中，PostgreSQL将向结果集注入三行。一行将被注入中东地区，一行将被注入北美地区。除此之外，我们还将得到一行总体平均数。如果我们正在建立一个网络应用，当前的结果是理想的，因为你可以很容易地建立一个GUI，通过过滤掉空值来钻取结果集。

当你想立即显示一个结果时，ROLLUP是合适的。就我个人而言，我一直用它来向终端用户显示最终结果。然而，如果你在做报告，那么你可能想预先计算更多的数据以确保更多的灵活性。CUBE关键字将帮助你做到这一点。

```
test=# SELECT region, country, avg(production)
 FROM t_oil
 WHERE country IN ('USA', 'Canada', 'Iran', 'Oman')
 GROUP BY CUBE (region, country);
 region | country | avg
---------------+---------+----------------------
 Middle East | Iran | 3631.6956521739130435
 Middle East | Oman | 586.4545454545454545
 Middle East | | 2142.9111111111111111
 North America | Canada | 2123.2173913043478261
 North America | USA | 9141.3478260869565217
 North America | | 5632.2826086956521739
 | | 3906.7692307692307692
 | Canada | 2123.2173913043478261
 | Iran | 3631.6956521739130435
 | Oman | 586.4545454545454545
 | USA | 9141.3478260869565217
(11 rows)
```

请注意，甚至更多的行已经被添加到结果中。CUBE 将创建与 GROUP BY 地区、国家 + GROUP BY 地区 + GROUP BY 国家 + 总体平均数相同的数据。所以，整个思路是一次性提取许多结果和各种级别的聚合。由此产生的立方体包含所有可能的分组组合。

ROLLUP和CUBE实际上只是在GROUPING SETS子句之上的便利功能。使用GROUPING SETS子句，你可以明确地列出你想要的聚合。

```
test=# SELECT region, country, avg(production)
 FROM t_oil
 WHERE country IN ('USA', 'Canada', 'Iran', 'Oman')
 GROUP BY GROUPING SETS ( (), region, country);
 region | country | avg
----------------+---------+----------------------
 Middle East | | 2142.9111111111111111
 North America | | 5632.2826086956521739
 | | 3906.7692307692307692
 | Canada | 2123.2173913043478261
 | Iran | 3631.6956521739130435
 | Oman | 586.4545454545454545
 | USA | 9141.3478260869565217
(7 rows)
```

在这一节中，我选择了三组分组：总体平均数、GROUP BY地区、GROUP BY国家。如果你想让地区和国家合并，就用（地区，国家）。

## 1.3 调查性能

分组集是一个强大的功能；它们有助于减少昂贵的查询次数。 在内部，PostgreSQL基本上会使用MixedAggregate来执行聚合。它可以同时执行许多操作，这保证了效率，如下例所示。

```
test=# explain SELECT region, country, avg(production)
 FROM t_oil
 WHERE country IN ('USA', 'Canada', 'Iran', 'Oman')
 GROUP BY GROUPING SETS ( (), region, country);
 QUERY PLAN
------------------------------------------------------------------------
 MixedAggregate (cost=0.00..18.17 rows=17 width=52)
 Hash Key: region
 Hash Key: country
 Group Key: ()
 -> Seq Scan on t_oil (cost=0.00..15.66 rows=184 width=24)
 Filter: (country = ANY ('{USA,Canada,Iran,Oman}'::text[]))
(6 rows)
```

在旧版本的PostgreSQL中，系统在所有情况下都使用GroupAggregate来执行这个操作。在更现代的版本中，已经添加了MixedAggregate。然而，你仍然可以使用enable_hashagg设置强迫优化器使用旧的策略。MixedAggregate本质上是HashAggregate，因此同样的设置也适用，如下例所示。

```
test=# SET enable_hashagg TO off;
SET
test=# explain SELECT region, country, avg(production)
 FROM t_oil
 WHERE country IN ('USA', 'Canada', 'Iran', 'Oman')
 GROUP BY GROUPING SETS ( (), region, country);
 QUERY PLAN
------------------------------------------------------------------------------
 GroupAggregate (cost=22.58..32.48 rows=17 width=52)
 Group Key: region
 Group Key: ()
 Sort Key: country
 Group Key: country
 -> Sort (cost=22.58..23.04 rows=184 width=24)
 Sort Key: region
 -> Seq Scan on t_oil (cost=0.00..15.66 rows=184 width=24)
 Filter: (country = ANY ('{USA,Canada,Iran,Oman}'::text[]))
(9 rows)
test=# SET enable_hashagg TO on;
SET
```

一般来说，基于哈希值的版本（MixedAggregate）速度更快，如果有足够的内存将MixedAggregate所需的哈希值保留在内存中，那么优化器就会倾向于这个版本。

## 1.4 将分组集与FILTER子句结合在一起

在现实世界的应用中，分组集经常可以和FILTER子句结合起来。 FILTER子句背后的想法是能够运行部分聚合。下面是一个例子。

```
test=# SELECT region,
 avg(production) AS all,
 avg(production) FILTER (WHERE year < 1990) AS old,
 avg(production) FILTER (WHERE year >= 1990) AS new
FROM t_oil
GROUP BY ROLLUP (region);
 region | all | old | new
--------------+----------------+----------------+----------------
 Middle East | 1992.603686635 | 1747.325892857 | 2254.233333333
 North America | 4541.362318840 | 4471.653333333 | 4624.349206349
 | 2607.513986013 | 2430.685618729 | 2801.183150183
(3 rows)
```

这里的想法是，不是所有的列都会使用相同的数据进行聚合。FILTER子句允许你有选择地将数据传递给这些聚合。在这个例子中，第二个聚合将只考虑1990年以前的数据，第三个聚合将照顾更多最近的数据，而第一个聚合将获得所有数据。

> 如果有可能将条件移到WHERE子句中，这总是比较理想的，因为需要从表中获取的数据较少。只有当WHERE子句留下的数据不被每个聚合所需要时，FILTER才有用。

FILTER适用于所有类型的聚合，并提供了一个简单的方法来透视你的数据。 此外，FILTER 比使用 CASE WHEN ... THEN NULL ... ELSE END 模拟相同行为更快。 你可以在这里找到一些真实的性能比较：https://www.cybertec-postgresql.com/en/postgresql-9-4aggregation-filters-they-do-pay-off/.

# 2 使用有序集

有序集是很强大的功能，但并没有被广泛地认为是这样，在开发者社区中也没有被广泛了解。这个想法其实很简单：数据被正常分组，然后在给定的条件下对每个组里面的数据进行排序。然后在这个排序的数据上进行计算。

一个经典的例子是中位数的计算

> 中位数是中间值。例如，如果你的收入是中位数，那么收入比你少和比你多的人的数量是相同的；50%的人做得更好，50%的人做得更糟。

获得中位数的一个方法是将排序后的数据移到数据集中的50%。这是一个关于WITHIN GROUP子句将要求PostgreSQL做什么的例子。

```
test=# SELECT region,
 percentile_disc(0.5) WITHIN GROUP (ORDER BY production)
FROM t_oil
GROUP BY 1;
 region | percentile_disc
----------------+-----------------
 Middle East | 1082
 North America | 3054
(2 rows)
```

percentile_disc函数将跳过50%的组别并返回所需的值。

> 请注意，中位数可能大大偏离平均值。

在经济学中，中位数和平均收入之间的偏差甚至可以作为社会平等或不平等的一个指标。

与平均数相比，中位数越高，收入不平等就越大。为了提供更多的灵活性，ANSI标准并没有仅仅提出一个中位数函数。相反，percentile_disc允许你使用0到1之间的任何数值。美妙之处在于，你甚至可以在使用有序集的同时使用分组集，如以下代码所示。

```
test=# SELECT region,
 percentile_disc(0.5) WITHIN GROUP (ORDER BY production)
FROM t_oil
GROUP BY ROLLUP (1);
 region | percentile_disc
----------------+-----------------
 Middle East | 1082
 North America | 3054
 | 1696
(3 rows)
```

在这种情况下，PostgreSQL将再次向结果集注入额外的行。

根据ANSI SQL标准的提议，PostgreSQL为你提供了两个percentile_函数。percentile_disc函数将返回一个数据集真正包含的值，而percentile_cont函数将在没有找到完全匹配的情况下插值。下面的例子显示了这是如何工作的。

```
test=# SELECT percentile_disc(0.62) WITHIN GROUP (ORDER BY id),
 percentile_cont(0.62) WITHIN GROUP (ORDER BY id)
FROM generate_series(1, 5) AS id;
 percentile_disc | percentile_cont
-----------------+-----------------
 4 | 3.48
(1 row)
```

4是一个真正存在的值-3.48已经被插值了。percentile_函数并不是PostgreSQL提供的唯一函数。为了找到一个组中最频繁的值，可以使用mode函数。在展示如何使用mode函数的例子之前，我已经编译了一个查询，告诉我们更多关于表的内容。

```
test=# SELECT production, count(*)
 FROM t_oil
 WHERE country = 'Other Middle East'
 GROUP BY production
 ORDER BY 2 DESC
 LIMIT 4;
 production | count
------------+-------
 50 | 5
 48 | 5
 52 | 5
 53 | 4
(4 rows)
```

三个不同的值正好出现五次。当然，模式函数只能给我们其中一个。

```
test=# SELECT country, mode() WITHIN GROUP (ORDER BY production)
 FROM t_oil
 WHERE country = 'Other Middle East'
 GROUP BY 1;
 country | mode
-------------------+------
 Other Middle East | 48
(1 row)
```

最频繁的值被返回，但是SQL不会告诉我们这个数字实际出现的频率。可能是这个数字只出现一次。

# 3 了解假设的聚合

假设的集合与标准的有序集合相当相似。然而，它们有助于回答一个不同的问题：如果一个值在数据中，结果会是什么？正如你所看到的，这不是关于数据库中的值，而是关于如果某个值确实存在的结果。

PostgreSQL提供的唯一的假设函数是等级，如下面的代码所示。

```
test=# SELECT region,
 rank(9000) WITHIN GROUP
 (ORDER BY production DESC NULLS LAST)
FROM t_oil
GROUP BY ROLLUP (1);
 region | rank
---------------+------
 Middle East | 21
 North America | 27
 | 47
(3 rows)
```

前面的代码告诉我们：如果有人每天生产9,000桶石油，它将在北美排名第27位，在中东排名第21位。

> 在这个例子中，我使用了NULLS LAST。当数据被排序时，空值通常在最后。然而，如果排序顺序被颠倒，空值仍然应该在列表的最后。NULLS LAST正是为了确保这一点。

# 4 利用窗口功能和分析

现在我们已经讨论了有序集合，现在是时候看看窗口函数了。 聚合遵循一个相当简单的原则：把许多行变成较少的聚合行。窗口函数则不同。它将当前行与该组中的所有行进行比较。返回的行数不会改变。下面是一个例子。

```
test=# SELECT avg(production) FROM t_oil;
 avg
-----------
 2607.5139
(1 row)
test=# SELECT country, year, production,
 consumption, avg(production) OVER ()
 FROM t_oil
 LIMIT 4;
country | year | production | consumption | avg
---------+-------+------------+-------------+----------
```

```
USA | 1965 | 9014 | 11522 | 2607.5139
USA | 1966 | 9579 | 12100 | 2607.5139
USA | 1967 | 10219 | 12567 | 2607.5139
USA | 1968 | 10600 | 13405 | 2607.5139
(4 rows)
```

在我们的数据集中，石油的平均产量约为260万桶/天。这个查询的目的是把这个值作为一个列加入。现在很容易将当前行与总体平均数进行比较。

请记住，OVER子句是必不可少的。如果没有这个子句，PostgreSQL就无法处理这个查询。

```
test=# SELECT country, year, production, consumption, avg(production) FROM
t_oil;
psql: ERROR: column "t_oil.country" must appear in the GROUP BY clause or be
used
 in an aggregate function
LINE 1: SELECT country, year, production, consumption, avg(productio...
```

这实际上是有意义的，因为平均数必须被精确地定义。数据库引擎不能只是猜测任何数值。

> 其他数据库引擎可以接受没有OVER或甚至GROUP BY子句的聚合函数。然而，从逻辑的角度来看，这是不对的，而且，还违反了SQL的规定。

## 4.1 分区数据

到目前为止，使用子选择也可以轻松实现同样的结果。然而，如果你想要的不仅仅是总体平均数，子选择会因为复杂而使你的查询变成噩梦。假设你不只是想要总体平均数，而是想要你所处理的国家的平均数。你需要的就是一个PARTITION BY子句。

```
test=# SELECT country, year, production, consumption,
 avg(production) OVER (PARTITION BY country)
FROM t_oil;
 country | year | production | consumption | avg
---------+------+------------+-------------+-----------
 Canada | 1965 | 920 | 1108 | 2123.2173
 Canada | 2010 | 3332 | 2316 | 2123.2173
 Canada | 2009 | 3202 | 2190 | 2123.2173
 ...
 Iran | 1966 | 2132 | 148 | 3631.6956
 Iran | 2010 | 4352 | 1874 | 3631.6956
 Iran | 2009 | 4249 | 2012 | 3631.6956
```

OVER子句定义了我们要看的窗口。在本例中，这个窗口是该行所属的国家。换句话说，该查询根据这个国家的其他行来返回行。

> 年份列没有被排序。该查询不包含明确的排序顺序，所以可能是数据以随机顺序返回。请记住，除非你明确说明你想要什么，否则SQL不承诺排序的输出。

基本上，PARTITION BY子句接受任何表达式。通常情况下，大多数人都会使用一个列来划分数据。下面是一个例子。

```
test=# SELECT year, production,
```

```
 avg(production) OVER (PARTITION BY year < 1990)
FROM t_oil
WHERE country = 'Canada'
ORDER BY year;
 year | production | avg
------+------------+----------------------
 1965 | 920 | 1631.6000000000000000
 1966 | 1012 | 1631.6000000000000000
 ...
 1990 | 1967 | 2708.4761904761904762
 1991 | 1983| 2708.4761904761904762
 1992 | 2065| 2708.4761904761904762
```

问题的关键是，数据是用表达式来分割的。year < 1990可以返回两个值：true或false。根据一个年份所在的组，它将被分配到1990年以前的平均值或1990年以后的平均值。PostgreSQL在这里真的很灵活。在现实世界的应用中，使用函数来确定组成员资格并不罕见。

## 4.2 窗口内的数据排序

PARTITION BY子句并不是唯一可以放在OVER子句中的东西。有时，有必要对窗口内的数据进行排序。ORDER BY将以某种方式向你的聚合函数提供数据。下面是一个例子。

```
test=# SELECT country, year, production,
 min(production) OVER (PARTITION BY country ORDER BY year)
FROM t_oil
WHERE year BETWEEN 1978 AND 1983
 AND country IN ('Iran', 'Oman');
country | year | production | min
---------+-----+------------+------
Iran | 1978 | 5302 | 5302
Iran | 1979 | 3218 | 3218
Iran | 1980 | 1479 | 1479
Iran | 1981 | 1321 | 1321
Iran | 1982 | 2397 | 1321
Iran | 1983 | 2454 | 1321
Oman | 1978 | 314 | 314
Oman | 1979 | 295 | 295
Oman | 1980 | 285 | 285
Oman | 1981 | 330 | 285
...
```

我们从数据集中选择了两个国家（伊朗和阿曼），时间为1978年至1983年。请记住，1979年伊朗正在进行革命，所以这对石油的生产有一定的影响。这些数据反映了这一点。

该查询所做的是计算到我们的时间序列中某一点的最低产量。在这一点上，让SQL学生记住ORDER BY子句在OVER子句中的作用是一个好方法。在这个例子中，PARTITION BY子句将为每个国家创建一个组，并在组内排序数据。min函数将循环处理排序后的数据，并提供所需的最小值。

如果你是窗口化函数的新手，有一点你应该注意。无论你是否使用ORDER BY子句，它确实会产生不同的效果。

```
test=# SELECT country, year, production,
 min(production) OVER (),
 min(production) OVER (ORDER BY year)
```

```
 FROM t_oil
 WHERE year BETWEEN 1978 AND 1983
 AND country = 'Iran';
 country | year | production | min | min
---------+------+------------+------+------
 Iran | 1978 | 5302| 1321 | 5302
 Iran | 1979 | 3218 | 1321 | 3218
 Iran | 1980 | 1479 | 1321 | 1479
 Iran | 1981 | 1321 | 1321 | 1321
 Iran | 1982 | 2397 | 1321 | 1321
 Iran | 1983 | 2454 | 1321 | 1321
(6 rows)
```

如果在没有ORDER BY的情况下使用聚合，它将会自动获取你窗口内整个数据集的最小值。如果有一个ORDER BY子句，这种情况就不会发生。在这种情况下，考虑到你所定义的顺序，它将始终是到此为止的最小值。

## 4.3 使用滑动窗口

到目前为止，我们在查询中使用的窗口是静态的。然而，对于像移动平均数这样的计算，这还不够。移动平均数需要一个滑动的窗口，随着数据的处理而移动。

下面是一个关于如何实现移动平均数的例子。

```
test=# SELECT country, year, production,
 min(production)
 OVER (PARTITION BY country
 ORDER BY year ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
FROM t_oil
WHERE year BETWEEN 1978 AND 1983
 AND country IN ('Iran', 'Oman');
country | year | production | min
---------+-------+------------+------
Iran | 1978 | 5302 | 3218
Iran | 1979 | 3218 | 1479
Iran | 1980 | 1479 | 1321
Iran | 1981 | 1321 | 1321
Iran | 1982 | 2397 | 1321
Iran | 1983 | 2454 | 2397
Oman | 1978 | 314 | 295
Oman | 1979 | 295 | 285
Oman | 1980 | 285 | 285
Oman | 1981 | 330 | 285
Oman | 1982 | 338 | 330
Oman | 1983 | 391 | 338
(12 rows)
```

最重要的是，移动窗口应该与ORDER BY子句一起使用。 否则，就会出现大问题。PostgreSQL实际上会接受这个查询，但结果会完全错误。记住，不先排序就把数据送入滑动窗口，只会导致随机数据。

ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING定义了窗口。在这个例子中，最多可以使用三行：当前行，之前的行，以及当前行之后的行。为了说明滑动窗口的工作原理，请看下面的例子。

```
test=# SELECT *, array_agg(id)
 OVER (ORDER BY id ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
 FROM generate_series(1, 5) AS id;
id | array_agg
----+-----------
 1 | {1,2}
 2 | {1,2,3}
 3 | {2,3,4}
 4 | {3,4,5}
 5 | {4,5}
(5 rows)
```

array_agg函数将把一个值的列表变成一个PostgreSQL数组。这将有助于解释滑动窗口的操作。

实际上，这个微不足道的查询有一些非常重要的方面。你可以看到的是，第一个数组只包含两个值。在1之前没有任何条目，因此数组不是满的。 PostgreSQL不会添加空条目，因为无论如何它们都会被聚合器忽略。 同样的情况也发生在数据的末端。

然而，滑动窗口提供了更多。有几个关键字可以用来指定滑动窗口。考虑一下下面的代码。

```
test=# SELECT *,
 array_agg(id) OVER (ORDER BY id ROWS BETWEEN
 UNBOUNDED PRECEDING AND 0 FOLLOWING)
FROM generate_series(1, 5) AS id;
 id | array_agg
----+-------------
 1 | {1}
 2 | {1,2}
 3 | {1,2,3}
 4 | {1,2,3,4}
 5 | {1,2,3,4,5}
(5 rows)
```

UNBOUNDED PRECEDING关键字意味着当前行之前的所有内容都会出现在窗口中。与UNBOUNDED PRECEDING相对应的是UNBOUNDED FOLLOWING。让我们看一下下面的例子。

```
test=# SELECT *,
 array_agg(id) OVER (ORDER BY id
 ROWS BETWEEN 2 FOLLOWING
 AND UNBOUNDED FOLLOWING)
FROM generate_series(1, 5) AS id;
 id | array_agg
----+-----------
 1 | {3,4,5}
 2 | {4,5}
 3 | {5}
 4 |
 5 |
(5 rows)
```

但还有一点：在某些情况下，你可能想把当前行从你的计算中排除。要做到这一点，SQL提供了一些语法糖，如下一个例子所示。

```
test=# SELECT year,
```

```
 production,
 array_agg(production) OVER (ORDER BY year
 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
EXCLUDE CURRENT ROW)
FROM t_oil
WHERE country = 'USA'
 AND year < 1970;
 year | production | array_agg
------+------------+---------------
 1965 | 9014 | {9579}
 1966 | 9579 | {9014,10219}
 1967 | 10219 | {9579,10600}
 1968 | 10600 | {10219,10828}
 1969 | 10828 | {10600}
(5 rows)
```

正如你所看到的，也可以使用一个在未来的窗口。PostgreSQL在这里是非常灵活的。

## 4.4 了解 ROWS 和 RANGE 之间的细微差别

到目前为止，你已经看到了使用OVER ...的滑动窗口。ROWS。然而，还有更多。让我们看看直接取自PostgreSQL文档的SQL规范。

```
{ RANGE | ROWS | GROUPS } frame_start [ frame_exclusion ]
{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end [ frame_exclusion ]
```

不仅仅是ROWS。在现实生活中，我们看到很多人都在努力理解RANGE和ROWS的区别。在很多情况下，结果都是一样的，这就更让人困惑了。为了理解这个问题，让我们首先创建一些简单的数据。

```
test=# SELECT *, x / 3 AS y FROM generate_series(1, 15) AS x;
 x | y
---+---
 1 | 0
 2 | 0
 3 | 1
 4 | 1
 5 | 1
 6 | 2
 7 | 2
 8 | 2
 9 | 3
 10 | 3
 11 | 3
 12 | 4
 13 | 4
 14 | 4
 15 | 5
(15 rows)
```

这是一个简单的数据集。要特别注意第二列，它包含几个重复的内容。它们在一分钟内就会有关系。

```
test=# SELECT *, x / 3 AS y,
 array_agg(x) OVER (ORDER BY x
 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS rows_1,
 array_agg(x) OVER (ORDER BY x
```

```
 RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS range_1,
 array_agg(x/3) OVER (ORDER BY (x/3)
 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS rows_2,
 array_agg(x/3) OVER (ORDER BY (x/3)
 RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS range_2
FROM generate_series(1, 15) AS x;
 x | y | rows_1 | range_1 | rows_2 | range_2
----+---+------------+------------+---------+---------------------
 1 | 0 | {1,2} | {1,2} | {0,0} | {0,0,1,1,1}
 2 | 0 | {1,2,3} | {1,2,3} | {0,0,1} | {0,0,1,1,1}
 3 | 1 | {2,3,4} | {2,3,4} | {0,1,1} | {0,0,1,1,1,2,2,2}
 4 | 1 | {3,4,5} | {3,4,5} | {1,1,1} | {0,0,1,1,1,2,2,2}
 5 | 1 | {4,5,6} | {4,5,6} | {1,1,2} | {0,0,1,1,1,2,2,2}
 6 | 2 | {5,6,7} | {5,6,7} | {1,2,2} | {1,1,1,2,2,2,3,3,3}
 7 | 2 | {6,7,8} | {6,7,8} | {2,2,2} | {1,1,1,2,2,2,3,3,3}
 8 | 2 | {7,8,9} | {7,8,9} | {2,2,3} | {1,1,1,2,2,2,3,3,3}
 9 | 3 | {8,9,10} | {8,9,10} | {2,3,3} | {2,2,2,3,3,3,4,4,4}
 10 | 3 | {9,10,11} | {9,10,11} | {3,3,3} | {2,2,2,3,3,3,4,4,4}
 11 | 3 | {10,11,12} | {10,11,12} | {3,3,4} | {2,2,2,3,3,3,4,4,4}
 12 | 4 | {11,12,13} | {11,12,13} | {3,4,4} | {3,3,3,4,4,4,5}
 13 | 4 | {12,13,14} | {12,13,14} | {4,4,4} | {3,3,3,4,4,4,5}
 14 | 4 | {13,14,15} | {13,14,15} | {4,4,5} | {3,3,3,4,4,4,5}
 15 | 5 | {14,15} | {14,15} | {4,5} | {4,4,4,5}
(15 rows)
```

在列出x和y列之后，我在x上应用了窗口函数。正如你所看到的，两列的结果是一样的。rows_1和range_1是绝对相同的。如果我们开始使用包含这些重复的列，情况就会改变。在ROWS的情况下，PostgreSQL只是简单地取上一行和下一行。在RANGE的情况下，它需要整个重复的组。因此，这个数组要长得多。整个一组相同的值都被拿走了。

## 4.5 使用 EXCLUDE TIES 和 EXCLUDE GROUP 删除重复项

有时，你想确保重复的内容不会进入窗口化函数的结果。EXCLUDE TIES子句正是帮助你实现这一目的。如果一个值在一个窗口中出现了两次，它将被删除。这是一个避免复杂的变通方法的好办法，因为变通方法可能既费钱又慢。下面的列表包含一个简单的例子。

```
SELECT *,
 x / 3 AS y,
 array_agg(x/3) OVER (ORDER BY x/3
 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS rows_1,
 array_agg(x/3) OVER (ORDER BY x/3
 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING EXCLUDE TIES) AS rows_2 FROM
generate_series(1, 10) AS x;
 x | y | rows_1 | rows_2
----+---+---------+--------
 1 | 0 | {0,0} | {0}
 2 | 0 | {0,0,1} | {0,1}
 3 | 1 | {0,1,1} | {0,1}
 4 | 1 | {1,1,1} | {1}
 5 | 1 | {1,1,2} | {1,2}
 6 | 2 | {1,2,2} | {1,2}
 7 | 2 | {2,2,2} | {2}
 8 | 2 | {2,2,3} | {2,3}
 9 | 3 | {2,3,3} | {2,3}
 10 | 3 | {3,3} | {3}
(10 rows)
```

我再次使用了generate_series函数来创建数据。使用一个简单的时间序列比挖掘一些更复杂的真实世界的数据要容易得多。 array_agg将把所有添加到窗口的值变成一个数组。然而，正如你在最后一栏中看到的，这个数组要短得多。重复的部分已经被自动删除。

除了EXCLUDE TIES子句之外，PostgreSQL还支持EXCLUDE GROUP。这里的意思是，你想在数据集进入聚合函数之前，从数据集中删除一整组记录。让我们看一下下面的例子。我们这里有四个窗口函数。第一个是你已经见过的经典的ROWS BETWEEN例子。我把这一列包括在内，以便更容易发现标准和EXCLUDE GROUP版本之间的差异。这里还需要注意的是，array_agg函数并不是你在这里唯一可以使用的函数--avg或任何其他窗口或聚合函数都可以正常工作。我只是用array_agg来让你更容易看到PostgreSQL的作用。在下面的例子中，你可以看到EXCLUDE GROUP删除了整组的记录。

```
SELECT *,
 x / 3 AS y,
 array_agg(x/3) OVER (ORDER BY x/3
 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS rows_1,
 avg(x/3) OVER (ORDER BY x/3
 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS avg_1,
 array_agg(x/3) OVER (ORDER BY x/3
 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING EXCLUDE GROUP) AS rows_2,
 avg(x/3) OVER (ORDER BY x/3
 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING EXCLUDE GROUP) AS avg_2
FROM generate_series(1, 10) AS x;
 x | y | rows_1 | avg_1 | rows_2 | avg_2
----+---+---------+----------+--------+----------
 1 | 0 | {0,0} | 0.000000 | |
 2 | 0 | {0,0,1} | 0.333333 | {1} | 1.000000
 3 | 1 | {0,1,1} | 0.666666 | {0} | 0.000000
 4 | 1 | {1,1,1} | 1.000000 | |
 5 | 1 | {1,1,2} | 1.333333 | {2} | 2.000000
 6 | 2 | {1,2,2} | 1.666666 | {1} | 1.000000
 7 | 2 | {2,2,2} | 2.000000 | |
 8 | 2 | {2,2,3} | 2.333333 | {3} | 3.000000
 9 | 3 | {2,3,3} | 2.666666 | {2} | 2.000000
 10 | 3 | {3,3} | 3.0000000 | |
(10 rows)
```

含有相同值的整个组被删除。当然，这也会影响到在这个结果之上计算的平均数。

## 4.6 抽象窗口子句

窗口函数允许我们向已计算好的结果集添加列。然而，很多列是基于同一个窗口的，这是一个经常出现的现象。把同样的子句反复放入你的查询中绝对不是一个好主意，因为你的查询会很难读懂，从而难以维护。

WINDOW子句允许开发人员预先定义一个窗口，并在查询中的不同地方使用它。它是这样工作的。

```
SELECT country, year, production,
 min(production) OVER (w),
 max(production) OVER (w)
FROM t_oil
WHERE country = 'Canada'
 AND year BETWEEN 1980
 AND 1985
WINDOW w AS (ORDER BY year);
country | year | production | min | max
```

```
--------+-------+------------+------+------
Canada | 1980 | 1764 | 1764 | 1764
Canada | 1981 | 1610 | 1610 | 1764
Canada | 1982 | 1590 | 1590 | 1764
Canada | 1983 | 1661 | 1590 | 1764
Canada | 1984 | 1775 | 1590 | 1775
Canada | 1985 | 1812 | 1590 | 1812
(6 rows)
```

前面的例子显示，min和max将使用同一个子句。 当然，可以有不止一个WINDOW子句--PostgreSQL在这里没有对用户施加严重的限制。

# 4.7 使用板载窗口功能

在向你介绍了基本概念之后，现在是时候看看PostgreSQL支持哪些开箱即用的窗口化函数了。你已经看到，窗口化与所有标准的聚合函数一起工作。在这些函数之上，PostgreSQL提供了一些窗口化和分析专用的额外函数。

在这一节中，我们将解释和讨论一些非常重要的函数。

## 4.7.1 rank 和 dense_rank 函数

根据我的判断，rank()和dense_rank()函数是SQL中最突出的函数。rank()函数返回当前行在其窗口中的编号。计数从1开始。

下面是一个例子。

```
test=# SELECT year, production,
 rank() OVER (ORDER BY production)
FROM t_oil
WHERE country = 'Other Middle East'
ORDER BY rank
LIMIT 7;
 year | production | rank
------+------------+------
 2001 | 47 | 1
 2004 | 48 | 2
 2002 | 48 | 2
 1999 | 48 | 2
 2000 | 48 | 2
 2003 | 48 | 2
 1998 | 49 | 7
(7 rows)
```

等级列将对你的数据集中的那些元祖进行编号。请注意，我的样本中的许多行是相等的。因此，等级将直接从2跳到7，因为许多产值是相同的。如果你想避免这种情况，dense_rank()函数才是解决这个问题的方法。

```
test=# SELECT year, production,
 dense_rank() OVER (ORDER BY production)
FROM t_oil
WHERE country = 'Other Middle East'
ORDER BY dense_rank
LIMIT 7;
```

```
 year | production | dense_rank
------+------------+------------
 2001 | 47 | 1
 2004 | 48 | 2
 ...
 2003 | 48 | 2
 1998 | 49 | 3
(7 rows)
```

PostgreSQL将把数字打包得更紧。将不会再有空隙。

## 4.7.2 ntile() 函数

一些应用程序需要将数据分成理想的相等的组。ntile()函数将完全为你做到这一点。

下面的例子显示了如何将数据分割成组。

```
test=# SELECT year, production,
 ntile(4) OVER (ORDER BY production)
 FROM t_oil
 WHERE country = 'Iraq'
 AND year BETWEEN 2000 AND 2006;
year | production | ntile
------+------------+-------
2003 | 1344 | 1
2005 | 1833 | 1
2006 | 1999 | 2
2004 | 2030 | 2
2002 | 2116 | 3
2001 | 2522 | 3
2000 | 2613 | 4
(7 rows)
```

该查询将数据分成四组。麻烦的是，只有七条记录被选中，这使得它不可能创建四个均匀的组。正如你所看到的，PostgreSQL将填满前三组，并使最后一组变得更小。你可以依靠这样一个事实：最后的组总是倾向于比其他组小一点。

> 在这个例子中，只使用了少量的行。在现实世界的应用中，将涉及数百万行，因此，如果分组不完全相等也没有问题。

ntile()函数通常不单独使用。当然，它有助于为某一行分配一个组的ID。 然而，在现实世界的应用中，人们希望在这些组之上进行计算。假设你想为你的数据创建一个四分位数分布。这就是它的工作原理。

```
test=# SELECT grp, min(production), max(production), count(*)
FROM (
 SELECT year, production,
 ntile(4) OVER (ORDER BY production) AS grp
 FROM t_oil
 WHERE country = 'Iraq'
 ) AS x
GROUP BY ROLLUP (1);
 grp | min | max | count
-----+------+------+-------
 1 | 285 | 1228 | 12
 2 | 1313 | 1977 | 12
 3 | 1999 | 2422 | 11
```

```
 4 | 2428 | 3489 | 11
 | 285 | 3489 | 46
(5 rows)
```

最重要的是，计算不能一步到位。当我在Cybertec（https://www.cybertec-postgresql.com）做SQL培训课程时，我试图向学生解释，只要你不知道如何一次完成，就考虑使用子选择。在分析学中，这通常是一个好主意。在这个例子中，在子选择中做的第一件事是给每个组附加一个组的标签。然后，这些组被取走并在主查询中处理。

其结果已经是可以在现实世界的应用中使用的了（例如，可能是作为位于图表旁边的一个图例）。

## 4.7.3 lead() 和 lag() 函数

ntile()函数对于将数据集分割成组是必不可少的，而lead()和lag()函数则是在结果集中移动线条。一个典型的用例是计算从一年到下一年的产量差异，如下面的例子所示。

```
test=# SELECT year, production,
 lag(production, 1) OVER (ORDER BY year)
 FROM t_oil
 WHERE country = 'Mexico'
 LIMIT 5;
 year | production | lag
------+------------+-----
 1965 | 362 |
 1966 | 370 | 362
 1967 | 411 | 370
 1968 | 439 | 411
 1969 | 461 | 439
(5 rows)
```

在实际计算产量的变化之前，坐下来看看lag()函数的实际作用是有意义的。你可以看到，该列被移动了一行。数据按照ORDER BY子句中的定义移动。在我的例子中，这意味着向下。当然，一个ORDER BY DESC子句会将数据向上移动。 从这一点上看，查询很容易。

```
test=# SELECT year, production,
 production - lag(production, 1) OVER (ORDER BY year)
 FROM t_oil
 WHERE country = 'Mexico'
 LIMIT 3;
 year | production | ?column?
------+------------+----------
 1965 | 362 |
 1966 | 370 | 8
 1967 | 411 | 41
(3 rows)
```

你所要做的就是像对待其他列那样计算差值。请注意，lag()函数有两个参数。第一个表示要显示的是哪一列。第二列告诉PostgreSQL你想移动多少行。那么，输入7，意味着一切都偏离了7行。 注意第一个值是Null（其他所有没有前面值的滞后行也是如此）。 lead()函数是滞后()函数的对应函数；它将向上而不是向下移动行。

```
test=# SELECT year, production,
 production - lead(production, 1) OVER (ORDER BY year)
 FROM t_oil
 WHERE country = 'Mexico'
 LIMIT 3;
 year | production | ?column?
------+------------+----------
 1965 | 362 | -8
 1966 | 370 | -41
 1967 | 411 | -28
(3 rows)
```

基本上，PostgreSQL也会接受前导列和滞后列的负值。因此，lag(production, -1)是前导(production, 1)的替代。然而，使用正确的函数将数据向你想要的方向移动肯定会更干净。

到目前为止，你已经看到了如何滞后一个单列。在大多数应用中，滞后一个值将是大多数开发人员使用的标准情况。关键是，PostgreSQL可以做的事情远不止这些。它可以滞后整个行。

```
test=# \x
Expanded display is on.
test=# SELECT year, production,
 lag(t_oil, 1) OVER (ORDER BY year)
 FROM t_oil
 WHERE country = 'USA'
 LIMIT 3;
-[ RECORD 1 ]-------------------------------------
year | 1965
production | 9014
lag |
-[ RECORD 2 ]-------------------------------------
year | 1966
production | 9579
lag | ("North America",USA,1965,9014,11522)
-[ RECORD 3 ]-------------------------------------
year | 1967
production | 10219
lag | ("North America",USA,1966,9579,12100)
```

这里的好处是，不仅仅是一个单一的值可以和前一行进行比较。但麻烦的是，PostgreSQL会把整个行作为一个复合类型返回，因此很难处理。要剖析一个复合类型，你可以使用括号和星号。

```
test=# SELECT year, production,
 (lag(t_oil, 1) OVER (ORDER BY year)).*
 FROM t_oil
 WHERE country = 'USA'
 LIMIT 3;
 year | prod | region | country | year | prod | consumption
------+------+------------+---------+------+------+-------------
 1965 | 9014 | | | | |
 1966 | 9579 | N. America | USA | 1965 | 9014 | 11522
 1967 | 10219 | N. America | USA | 1966 | 9579 | 12100
(3 rows)
```

为什么会有这样的作用？滞后整个行将使我们有可能看到数据是否被插入了不止一次。在你的时间序列数据中检测重复的行（或接近重复的行）是非常简单的。

请看下面的例子。

```
test=# SELECT *
 FROM (SELECT t_oil, lag(t_oil) OVER (ORDER BY year)
 FROM t_oil
 WHERE country = 'USA'
 ) AS x
 WHERE t_oil = lag;
 t_oil | lag
--------+-----
(0 rows)
```

当然，样本数据并不包含重复的内容。然而，在现实世界的例子中，重复很容易发生，即使没有主键，也很容易检测到它们。

> t_oil行实际上是整个行。由子选择返回的滞后也是一个完整的行。在PostgreSQL中，在字段相同的情况下，复合类型可以直接进行比较。PostgreSQL将简单地在一个字段之后进行比较。

## 4.7.4 first_value(), nth_value(), and last_value()函数

有时，有必要根据数据窗口的第一个值来计算数据。 不出所料，这样做的函数是first_value()。

```
test=# SELECT year, production,
 first_value(production) OVER (ORDER BY year)
 FROM t_oil
 WHERE country = 'Canada'
 LIMIT 4;
 year | production | first_value
-------+------------+-------------
 1965 | 920 | 920
 1966 | 1012 | 920
 1967 | 1106 | 920
 1968 | 1194 | 920
(4 rows)
```

同样，需要一个排序顺序来告诉系统第一个值的实际位置。然后PostgreSQL会把同样的值放到最后一列。如果你想找到窗口中的最后一个值，只需使用last_value()函数而不是first_value()函数。

如果你对第一个或最后一个值不感兴趣，而是要寻找中间的东西，PostgreSQL提供了nth_value()函数。

```
test=# SELECT year, production,
 nth_value(production, 3) OVER (ORDER BY year)
 FROM t_oil
 WHERE country = 'Canada';
 year | production | nth_value
-------+------------+-----------
 1965 | 920 |
 1966 | 1012 |
 1967 | 1106 | 1106
 1968 | 1194 | 1106
 ...
```

在这种情况下，第三个值将被放入最后一列。然而，请注意，前两行是空的。问题是，当PostgreSQL开始浏览数据时，第三个值还不知道。因此，空值被加入。现在的问题是，我们怎样才能使时间序列更加完整，用即将到来的数据替换那两个空值呢？这里有一个方法。

```
test=# SELECT *, min(nth_value) OVER ()
FROM (
 SELECT year, production,
 nth_value(production, 3) OVER (ORDER BY year)
 FROM t_oil
 WHERE country = 'Canada'
 ) AS x
LIMIT 4;
 year | production | nth_value | min
------+------------+-----------+------
 1965 | 920 | | 1106
 1966 | 1012 | | 1106
 1967 | 1106 | 1106 | 1106
 1968 | 1194 | 1106 | 1106
(4 rows)
```

子选择将创建一个不完整的时间序列。在此之上的SELECT子句将完成数据。这里的线索是，完成数据可能会更复杂，因此子选择可能会创造一些机会来添加更复杂的逻辑，而不是只在一个步骤中完成。

## 4.7.5 row_number() 函数

本节要讨论的最后一个函数是row_number()函数，它可以简单地用来返回一个虚拟ID。听起来很简单，不是吗？在这里，它是。

```
test=# SELECT country, production,
 row_number() OVER (ORDER BY production)
 FROM t_oil
 LIMIT 3;
 country | production | row_number
---------+------------+------------
 Yemen | 10 | 1
 Syria | 21 | 2
 Yemen | 26 | 3
(3 rows)
```

row_number()函数只是为该行分配了一个数字。肯定不会有重复的。这里有趣的一点是，即使没有订单也能做到这一点（如果它与您无关）。

```
test=# SELECT country, production,
 row_number() OVER()
 FROM t_oil
 LIMIT 3;
 country | production | row_number
---------+------------+------------
 USA | 9014 | 1
 USA | 9579 | 2
 USA | 10219 | 3
(3 rows)
```

结果正是我们所期望的。

# 5 编写你自己的聚合

在本书中，你将了解到PostgreSQL提供的大部分内置函数。 然而，SQL提供的东西可能对你来说还不够。好消息是，有可能在数据库引擎中添加你自己的聚合函数。在本节中，你将学习如何做到这一点。

## 5.1 创建简单聚合

对于这个例子，目标是解决一个非常简单的问题。如果顾客乘坐出租车，他们通常要为上车付费--例如，2.5欧元。现在，我们假设每走一公里，顾客需要支付2.2欧元。现在的问题是，一次旅行的总价格是多少?

当然，这个例子很简单，不需要自定义聚合就可以解决；但是，让我们看看它是如何工作的。首先，需要创建一些测试数据。

```
test=# CREATE TABLE t_taxi (trip_id int, km numeric);
CREATE TABLE
test=# INSERT INTO t_taxi
 VALUES (1, 4.0), (1, 3.2), (1, 4.5), (2, 1.9), (2, 4.5);
INSERT 0 5
```

为了创建聚合，PostgreSQL提供了CREATE AGGREGATE命令。这个命令的语法随着时间的推移已经变得非常强大和冗长，以至于在本书中包括它的输出已经没有意义了。相反，我建议去看PostgreSQL的文档，可以在https://www.postgresql.org/docs/devel/static/sql-createaggregate.html找到。 编写聚合时，首先需要的是一个函数，每一行都要调用这个函数。它将接受一个中间值，以及来自被处理行的数据。 下面是一个例子。

```
test=# CREATE FUNCTION taxi_per_line (numeric, numeric)
RETURNS numeric AS
$$
 BEGIN
 RAISE NOTICE 'intermediate: %, per row: %', $1, $2;
 RETURN $1 + $2*2.2;
 END;
$$
LANGUAGE 'plpgsql';
CREATE FUNCTION
```

现在，已经可以创建一个简单的聚合：

```
test=# CREATE AGGREGATE taxi_price (numeric)
(
 INITCOND = 2.5,
 SFUNC = taxi_per_line,
 STYPE = numeric
);
CREATE AGGREGATE
```

正如我们之前所说的，每一次旅行都以2.5欧元开始，用于上出租车，这是由INITCOND（初始条件）定义的。它代表每组的起始值。然后，为组内的每条线路调用一个函数。在我的例子中，这个函数是taxi_per_line，已经被定义了。正如你所看到的，它需要两个参数。第一个参数是一个中间值。那些额外的参数是由用户传递给函数的参数。

下面的语句显示了哪些数据被传递，何时传递，以及如何传递。

```
test=# SELECT trip_id, taxi_price(km) FROM t_taxi GROUP BY 1;
psql: NOTICE: intermediate: 2.5, per row: 4.0
psql: NOTICE: intermediate: 11.30, per row: 3.2
psql: NOTICE: intermediate: 18.34, per row: 4.5
psql: NOTICE: intermediate: 2.5, per row: 1.9
psql: NOTICE: intermediate: 6.68, per row: 4.5
psql: trip_id | taxi_price
---------+------------
 1 | 28.24
 2 | 16.58
(2 rows)
```

该系统从行程1和2.50欧元（初始条件）开始。然后，增加4公里。 总的来说，现在的价格是2.50 + 4 x 2.2。然后，加入下一条线路，将增加3.2 x 2.2，以此类推。因此，第一次旅行的费用为28.24欧元。

然后，下一次旅行开始。同样，有一个新的启动条件，PostgreSQL将每行调用一个函数。

在PostgreSQL中，一个集合体也可以自动作为一个窗口函数使用。不需要额外的步骤-你可以直接使用聚合。

```
test=# SELECT *, taxi_price(km) OVER (PARTITION BY trip_id ORDER BY km)
 FROM t_taxi;
psql: NOTICE: intermediate: 2.5, per row: 3.2
psql: NOTICE: intermediate: 9.54, per row: 4.0
psql: NOTICE: intermediate: 18.34, per row: 4.5
psql: NOTICE: intermediate: 2.5, per row: 1.9
psql: NOTICE: intermediate: 6.68, per row: 4.5
 trip_id | km | taxi_price
---------+-----+------------
 1 | 3.2 | 9.54
 1 | 4.0 | 18.34
 1 | 4.5 | 28.24
 2 | 1.9 | 6.68
 2 | 4.5 | 16.58
(5 rows)
```

该查询所做的是给我们提供到行程中某一点的价格。

我们定义的聚合将在每行调用一个函数。然而，用户如何能够计算出一个平均数呢？如果不添加FINALFUNC函数，这样的计算是不可能的。为了演示FINALFUNC是如何工作的，我们必须扩展我们的例子。假设顾客想在离开出租车后立即给出租车司机10%的小费。这10%必须在最后加上，一旦知道了总价，就必须加上。这就是FINALFUNC发挥作用的地方。它是这样工作的。

```
test=# DROP AGGREGATE taxi_price(numeric);
DROP AGGREGATE
```

首先，旧的聚合体被放弃。然后，FINALFUNC被定义。它将获得中间结果作为一个参数，并施展它的魔力。

```
test=# CREATE FUNCTION taxi_final (numeric)
 RETURNS numeric AS
$$
 SELECT $1 * 1.1;
$$
LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
```

在这种情况下，计算是非常简单的--正如我们之前所说，10%被添加到最终的总和中。 一旦该函数被部署，就已经可以重新创建总量了。

```
test=# CREATE AGGREGATE taxi_price (numeric)
(
 INITCOND = 2.5,
 SFUNC = taxi_per_line,
 STYPE = numeric,
 FINALFUNC = taxi_final
);
CREATE AGGREGATE
```

最后，价格只会比以前高一点：

```
test=# SELECT trip_id, taxi_price(km) FROM t_taxi GROUP BY 1;
psql: NOTICE: intermediate: 2.5, per row: 4.0
...
 trip_id | taxi_price
---------+------------
 1 | 31.064
 2 | 18.238
(2 rows)
```

PostgreSQL会自动处理所有的分组等问题。

对于简单的计算，可以使用简单的数据类型作为中间结果。然而，并不是所有的操作都可以通过仅仅传递简单的数字和文本来完成。幸运的是，PostgreSQL允许使用复合数据类型，它可以作为中间结果使用。

想象一下，你想计算一些数据的平均值，也许是一个时间序列。一个中间结果可能看起来如下。

```
test=# CREATE TYPE my_intermediate AS (c int4, s numeric);
CREATE TYPE
```

可以自由地组成任何符合你的目的的任意类型。只要把它作为第一个参数传递，并根据需要添加数据作为附加参数。

## 5.2 添加对并行查询的支持

你刚才看到的是一个简单的聚合，它不支持并行查询等。为了解决这些难题，下面的几个例子都是关于改进和加速的。

在创建一个聚合时，你可以选择性地定义以下内容。

```
[ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
```

默认情况下，一个聚合体不支持并行查询。然而，出于性能方面的考虑，明确说明聚合的能力是有意义的。

- UNSAFE:在这种模式下，不允许并行查询。
- RESTRICTED:在这种模式下，可以在并行模式下执行集合，但执行仅限于并行组长。
- SAFE在这种模式下，它提供对并行查询的全面支持。

如果你把一个函数标记为SAFE，你必须牢记，这个函数不能有副作用。执行顺序不能对查询的结果产生影响。只有这样，PostgreSQL才能被允许并行地执行操作。没有副作用的函数的例子是 sin(x) 和 length(s)。IMMUTABLE函数是很好的候选函数，因为它们保证在相同的输入下返回相同的结果。 如果有某些限制的话，STABLE函数可以工作。

## 5.3 提高效率

到目前为止，我们所定义的聚合体已经可以实现相当多的功能。然而，如果你使用的是滑动窗口，那么函数调用的数量将直接爆炸。这就是所发生的事情。

```
test=# SELECT taxi_price(x::numeric)
 OVER (ROWS BETWEEN 0 FOLLOWING AND 3 FOLLOWING)
FROM generate_series(1, 5) AS x;
psql: NOTICE: intermediate: 2.5, per row: 1
psql: NOTICE: intermediate: 4.7, per row: 2
psql: NOTICE: intermediate: 9.1, per row: 3
psql: NOTICE: intermediate: 15.7, per row: 4
psql: NOTICE: intermediate: 2.5, per row: 2
psql: NOTICE: intermediate: 6.9, per row: 3
psql: NOTICE: intermediate: 13.5, per row: 4
psql: NOTICE: intermediate: 22.3, per row: 5
...
```

对于每一行，PostgreSQL将处理整个窗口。如果滑动窗口很大，效率就会下降。为了解决这个问题，我们的聚合体可以被扩展。在此之前，旧的聚合体可以被放弃。

```
DROP AGGREGATE taxi_price(numeric);
```

基本上，需要两个函数。msfunc函数将把窗口中的下一行添加到中间结果中。

```
CREATE FUNCTION taxi_msfunc(numeric, numeric)
 RETURNS numeric AS
$$
 BEGIN
 RAISE NOTICE 'taxi_msfunc called with % and %', $1, $2;
 RETURN $1 + $2;
 END;
$$ LANGUAGE 'plpgsql' STRICT;
```

minvfunc 函数将从中间结果中移除掉落在窗口之外的值：

```
CREATE FUNCTION taxi_minvfunc(numeric, numeric) RETURNS numeric AS
$$
BEGIN
 RAISE NOTICE 'taxi_minvfunc called with % and %', $1, $2;
 RETURN $1 - $2;
END;
$$
LANGUAGE 'plpgsql' STRICT;
```

在这个例子中，我们所做的只是加和减。在一个更复杂的例子中，计算可以是任意复杂的。

下面的语句显示了如何重新创建聚合的情况

```
CREATE AGGREGATE taxi_price (numeric)
(
 INITCOND = 0,
 STYPE = numeric,
 SFUNC = taxi_per_line,
 MSFUNC = taxi_msfunc,
 MINVFUNC = taxi_minvfunc,
 MSTYPE = numeric
);
```

现在让我们再次运行相同的查询：

```
test# SELECT taxi_price(x::numeric)
 OVER (ROWS BETWEEN 0 FOLLOWING AND 3 FOLLOWING)
FROM generate_series(1, 5) AS x;
psql: NOTICE: taxi_msfunc called with 1 and 2
psql: NOTICE: taxi_msfunc called with 3 and 3
psql: NOTICE: taxi_msfunc called with 6 and 4
psql: NOTICE: taxi_minfunc called with 10 and 1
psql: NOTICE: taxi_msfunc called with 9 and 5
psql: NOTICE: taxi_minfunc called with 14 and 2
psql: NOTICE: taxi_minfunc called with 12 and 3
psql: NOTICE: taxi_minfunc called with 9 and 4
```

函数调用的数量急剧减少。每行只需执行固定的几个调用。不再需要重新计算同一个框架。

## 5.4 编写假设聚合

编写聚合并不难，而且对于执行更复杂的操作非常有利。在本节中，计划编写一个假想的聚合体，本章已经讨论过了。

实现假设的聚合体与编写普通的聚合体没有太大区别。真正困难的部分是弄清楚什么时候要真正使用一个。为了使这一节尽可能容易理解，我决定包括一个微不足道的例子：给定一个特定的顺序，如果我们在字符串的末尾加上abc，结果会是什么？

它是这样工作的：

```
CREATE AGGREGATE name ( [ [ argmode ] [ argname ] arg_data_type [ , ... ] ]
 ORDER BY [ argmode ] [ argname ] arg_data_type
 [ , ...])
(
 SFUNC = sfunc,
 STYPE = state_data_type
 [ , SSPACE = state_data_size ]
 [ , FINALFUNC = ffunc ]
 [ , FINALFUNC_EXTRA ]
 [ , INITCOND = initial_condition ]
 [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ] [ , HYPOTHETICAL ]
)
```

将需要两个函数：sfunc和finalfunc。sfunc函数将为每一行调用。

```
CREATE FUNCTION hypo_sfunc(text, text)
 RETURNS text AS
$$
 BEGIN
 RAISE NOTICE 'hypo_sfunc called with % and %', $1, $2;
 RETURN $1 || $2;
 END;
$$ LANGUAGE 'plpgsql';
```

两个文本参数将被传递给该过程。我们用它们来进行连接。其逻辑与之前的相同。就像我们之前做的那样，可以定义一个最终的函数调用。

```
CREATE FUNCTION hypo_final(text, text, text)
 RETURNS text AS
$$
 BEGIN
 RAISE NOTICE 'hypo_final called with %, %, and %',
 $1, $2, $3;
 RETURN $1 || $2;
 END;
$$ LANGUAGE 'plpgsql';
```

一旦这些功能到位，就可以创建假想的聚合。

```
CREATE AGGREGATE whatif(text ORDER BY text)
(
 INITCOND = 'START',
 STYPE = text,
 SFUNC = hypo_sfunc,
 FINALFUNC = hypo_final,
 FINALFUNC_EXTRA = true,
 HYPOTHETICAL
);
```

请注意，这个聚合已经被标记为假想的，以便PostgreSQL知道它实际上是什么样的集合。 现在聚合已经创建，可以运行它了

```
test=# SELECT whatif('abc'::text) WITHIN GROUP (ORDER BY id::text)
 FROM generate_series(1, 3) AS id;
psql: NOTICE: hypo_sfunc called with START and 1
psql: NOTICE: hypo_sfunc called with START1 and 2
psql: NOTICE: hypo_sfunc called with START12 and 3
psql: NOTICE: hypo_final called with START123, abc, and <NULL>
 whatif
-------------
 START123abc
(1 row)
```

理解所有这些集合体的关键是要充分看到每一种函数被调用时会发生什么，以及整个机器是如何工作的。

# 6 总结

在本章中，你了解了SQL提供的高级功能。在简单的聚合之上，PostgreSQL提供了有序集、分组集、窗口函数和递归，以及一个你可以用来创建自定义聚合的接口。在数据库中运行聚合的好处是，代码很容易编写，而且在效率方面，数据库引擎通常会有优势。

在第5章 "日志文件和系统统计 "中，我们将把注意力转向更多的管理任务，如处理日志文件、了解系统统计和实施监控等。