优化查询以获得良好的性能

在第5章 "日志文件和系统统计 "中，你学会了如何阅读系统统计，以及如何利用PostgreSQL提供的内容。现在我们已经掌握了这些知识，这一章的重点是良好的查询性能。每个人都在寻求良好的查询性能。因此，以深入的方式处理这个话题是很重要的。

在本章中，你将了解到以下内容。

- 学习优化器的作用
- 了解执行计划
- 了解和修复连接
- 启用和禁用优化器设置
- 分区数据
- 调整参数以获得良好的查询性能
- 利用并行查询
- 引入即时编译（JIT）

在本章结束时，我们将能够写出更好、更快的查询。如果查询的结果仍然不是很好，我们应该能够理解为什么会出现这种情况。我们还将能够使用我们将学到的新技术来划分数据。

1 学习优化器的作用

在尝试考虑查询性能之前，熟悉一下查询优化器的工作是有意义的。深入了解引擎盖下发生的事情是很有意义的，因为它可以帮助你看到数据库真正在做什么。

1.1 一个实际的例子--查询优化器如何处理一个样本查询

为了演示优化器是如何工作的，我编译了一个例子。这是我多年来在PostgreSQL培训中使用的东西。让我们假设有三个表，如下所示。

```
CREATE TABLE a (aid int, ...); -- 100 million rows
CREATE TABLE b (bid int, ...); -- 200 million rows
CREATE TABLE c (cid int, ...); -- 300 million rows
```

让我们进一步假设，这些表包含了数百万，甚至上亿的行。除此以外，还有索引。

```
CREATE INDEX idx_a ON a (aid);
CREATE INDEX idx_b ON b (bid);
CREATE INDEX idx_c ON c (cid);
CREATE VIEW v AS SELECT *
 FROM a, b
 WHERE aid = bid;
```

最后，有一个视图将前两个表连接在一起。 让我们假设最终用户想运行以下查询。优化器会对这个查询做什么？计划员有什么选择？

```
SELECT *
FROM v, c
WHERE v.aid = c.cid
 AND cid = 4;
```

在研究实际的优化过程之前，我们将重点讨论计划者拥有的一些选项。

1.2 评估连接方案

规划者在这里有几个选择，所以让我们借此机会了解一下如果使用直接的方法会出什么问题。

假设计划员只是稳步前进，计算视图的输出。将1亿条记录与2亿条记录连接起来的最佳方法是什么？

在这一节中，将讨论几个（不是全部）连接选项，向你展示PostgreSQL能够做什么。

1.2.1 嵌套循环 Nested loops

连接两个表的一种方法是使用一个嵌套循环。这里的原理很简单。下面是一些伪代码。

```
for x in table1:
 for y in table2:
 if x.field == y.field
 issue row
 else
 keep doing
```

如果其中一方非常小，并且只包含有限的数据集，则经常使用嵌套循环。在我们的例子中，一个嵌套循环会导致1亿x2亿的代码迭代。这显然不是一个选项，因为运行时间会直接爆炸。

嵌套循环通常是O(n2)，所以只有在连接的一方非常小的情况下，它才有效。在这个例子中，情况并非如此，因此可以排除嵌套循环来计算视图

1.2.2 哈希连接 Hash joins

第二个选择是哈希连接。可以应用以下策略来解决我们的小问题。 下面的列表显示了散列连接是如何工作的。

```
Hash join
 Sequential scan table 1
 Sequential scan table 2
```

两边都可以被哈希化，哈希键可以被比较，留给我们的是连接的结果。 这里的问题是，所有的值都必须被散列并存储在某个地方。

1.2.3 合并联接 Merge joins

最后，是合并连接。这里的想法是使用排序的列表来连接结果。如果连接的两边都是排序的，系统就可以从最上面的行中抽取，看它们是否匹配并返回。这里的主要要求是，列表是排序的。下面是一个计划样本。

```
Merge join
 Sort table 1
 Sequential scan table 1
 Sort table 2
 Sequential scan table 2
```

为了连接这两个表（表1和表2），必须以分类的顺序提供数据。在许多情况下，PostgreSQL将只是对数据进行排序。然而，我们还可以使用其他的选项来为连接提供分类的数据。一种方法是查阅一个索引，如下面的例子所示。

```
Merge join
 Index scan table 1
 Index scan table 2
```

连接的一边，或者两边，可以使用来自计划中较低层次的排序数据。如果直接访问表，索引是明显的选择，但是只有在返回的结果集明显小于整个表的情况下。否则，我们会遇到几乎双倍的开销，因为我们必须读取整个索引，然后再读取整个表。如果结果集是表的很大一部分，那么顺序扫描会更有效率，特别是在以主键顺序访问时。

合并连接的优点是它可以处理大量的数据。缺点是，在某些时候必须对数据进行排序或从索引中提取。

排序是O(n * log(n))。因此，对3亿行进行排序以执行连接也是没有吸引力的。

> 请注意，自从引入PostgreSQL 10.0后，这里描述的所有连接选项也可以在并行版本中使用。因此，优化器不会只考虑那些标准的连接选项，也会评估执行并行查询是否有意义。

1.3 应用转换

很明显，做明显的事情（先加入视图）是完全没有意义的。嵌套循环会使执行时间超过屋顶。哈希连接必须对数百万条记录进行哈希处理，而嵌套循环必须对3亿条记录进行排序。这三个选项显然都不适合这里。出路是应用逻辑转换来使查询快速。在这一节中，你将学习规划器为加快查询速度所做的工作。有几个步骤需要执行。

- 内联视图:优化器所做的第一个转换是内联视图。下面是发生的情况。

```
SELECT *
FROM
(
 SELECT *
 FROM a, b
 WHERE aid = bid
) AS v, c
WHERE v.aid = c.cid
 AND cid = 4;
```

视图被内联并转化为一个子选择。这对我们有什么好处？实际上，什么都没有。它所做的只是为进一步优化打开了大门，这将真正改变这个查询的游戏规则。

- 扁平化子选择:我们需要做的下一件事是扁平化子选择，这意味着将它们整合到主查询中。通过摆脱子选择，会出现更多的选项，我们可以利用这些选项来优化查询。

下面是扁平化子选择后的查询的情况。

```
SELECT * FROM a, b, c WHERE a.aid = c.cid AND aid = bid AND cid = 4;
```

现在，它是一个正常的连接，提供了应用更多优化的选项。如果没有这个内联步骤，这将是不可能的。让我们看看现在有哪些优化措施可用。

> 我们可以自己重写这个SQL，但无论如何，计划器将为我们处理这些转换。优化器现在可以进行进一步的优化。

1.4 应用等式约束

下面的过程创建了平等约束。这个想法是为了检测额外的约束、连接选项和过滤器。让我们深呼吸一下，看看下面的查询：如果aid=cid，aid=bid，我们知道bid=cid。如果cid=4，其他的都一样，我们知道aid和bid也必须是4，这就导致了下面的查询。

```
SELECT *
FROM a, b, c
WHERE a.aid = c.cid
 AND aid = bid
 AND cid = 4
 AND bid = cid
 AND aid = 4
 AND bid = 4
```

这个优化的重要性怎么强调都不为过。计划员在这里所做的是为两个额外的索引打开了大门，这些索引在原始查询中并不明显。 由于能够在所有三列上使用索引，现在的查询就便宜多了。优化器可以选择只从索引中检索几条记录，并使用任何有意义的连接选项。

1.5 详尽的搜索

现在这些形式上的转换已经完成了，PostgreSQL将进行一次详尽的搜索。它将尝试所有可能的计划，并为你的查询提出最便宜的解决方案。 PostgreSQL知道哪些索引是可能的，只是使用成本模型来确定如何以最好的方式来做事情。

在穷举搜索期间，PostgreSQL也会尝试确定最佳的连接顺序。在最初的查询中，连接顺序被固定为A→B和A→C。然而，使用那些平等约束，我们可以连接B→C，然后再连接A。所有这些选项对规划者都是开放的。

1.6 检查执行计划

现在已经讨论了所有的优化选项，现在是时候看看PostgreSQL会产生什么样的执行计划了。 让我们先试着用完全分析过的空表来进行查询。

```
test=# explain SELECT * FROM v, c WHERE v.aid = c.cid AND cid = 4;
 QUERY PLAN
------------------------------------------------------------------------------
------
 Nested Loop (cost=12.77..74.50 rows=2197 width=12)
 -> Nested Loop (cost=8.51..32.05 rows=169 width=8)
 -> Bitmap Heap Scan on a (cost=4.26..14.95 rows=13 width=4)
 Recheck Cond: (aid = 4)
 -> Bitmap Index Scan on idx_a (cost=0.00..4.25 rows=13 width=0)
 Index Cond: (aid = 4)
 -> Materialize (cost=4.26..15.02 rows=13 width=4)
 -> Bitmap Heap Scan on b (cost=4.26..14.95 rows=13 width=4)
 Recheck Cond: (bid = 4)
-> Bitmap Index Scan on idx_b (cost=0.00..4.25 rows=13 width=0)
 Index Cond: (bid = 4)
 -> Materialize (cost=4.26..15.02 rows=13 width=4)
 -> Bitmap Heap Scan on c (cost=4.26..14.95 rows=13 width=4)
 Recheck Cond: (cid = 4)
 -> Bitmap Index Scan on idx_c (cost=0.00..4.25 rows=13 width=0)
 Index Cond: (cid = 4)
(16 rows)
```

你所看到的是使用空表产生的计划。然而，让我们看看如果我们添加数据会发生什么。

```
test=# INSERT INTO a SELECT * FROM generate_series(1, 1000000);
INSERT 0 1000000
test=# INSERT INTO b SELECT * FROM generate_series(1, 1000000);
INSERT 0 1000000
test=# INSERT INTO c SELECT * FROM generate_series(1, 1000000);
INSERT 0 1000000
test=# ANALYZE ;
ANALYZE
```

如下面的代码所示，计划已经改变。然而，重要的是，在这两个计划中，你会看到过滤器被自动应用于查询中的所有列。PostgreSQL的平等约束已经完成了它们的工作。

```
test=# explain SELECT * FROM v, c WHERE v.aid = c.cid AND cid = 4;
 QUERY PLAN
------------------------------------------------------------------------------------
 Nested Loop (cost=1.27..13.35 rows=1 width=12)
 -> Nested Loop (cost=0.85..8.89 rows=1 width=8)
 -> Index Only Scan using idx_a on a (cost=0.42..4.44 rows=1 width=4)
 Index Cond: (aid = 4)
 -> Index Only Scan using idx_b on b (cost=0.42..4.44 rows=1 width=4)
 Index Cond: (bid = 4)
 -> Index Only Scan using idx_c on c (cost=0.42..4.44 rows=1 width=4)
 Index Cond: (cid = 4)
(8 rows)
```

> 请注意，本章中显示的计划不一定与你将观察到的情况100%相同。取决于你装载了多少数据，可能会有轻微的变化。成本也可能取决于磁盘上数据的物理排列（磁盘上的顺序）。在运行这些例子时，请牢记这一点。

正如你所看到的，PostgreSQL将使用三个索引。同样有趣的是，PostgreSQL决定采用一个嵌套循环来连接数据。这很有意义，因为几乎没有从索引扫描回来的数据。因此，使用一个循环来连接东西是完全可行的，而且效率很高。

1.7 使过程失败

到目前为止，你已经看到了PostgreSQL可以为你做什么，以及优化器如何帮助加快查询速度。PostgreSQL是相当聪明的，但它需要聪明的用户。在有些情况下，终端用户会因为做一些愚蠢的事情而使整个优化过程陷入瘫痪。让我们通过使用以下命令来放弃视图。

```
test=# DROP VIEW v;
DROP VIEW
```

现在，该视图已经被重新创建了。请注意，OFFSET 0已经被添加到视图的末端。让我们看一下下面的例子。

```
test=# CREATE VIEW v AS SELECT *
 FROM a, b
 WHERE aid = bid
 OFFSET 0;
CREATE VIEW
```

虽然这个视图在逻辑上等同于之前展示的例子，但优化器必须以不同的方式处理事情。除了0以外的每一个OFFSET都会改变结果，因此必须对该视图进行计算。整个优化过程因为加入了OFFSET这样的东西而变得残缺不全。

> PostgreSQL社区已经决定不优化这种模式。如果你在视图中使用OFFSET 0，计划器就不会把它剥离出来。人们根本不应该这么做。我们将用这个例子来观察某些操作是如何削弱性能的，而且我们作为开发者，应该意识到潜在的优化过程。 然而，如果你碰巧知道PostgreSQL是如何工作的，这一招就可以用于优化。

如果视图包含 OFFSET 0，则这是新计划：

```
test=# EXPLAIN SELECT * FROM v, c WHERE v.aid = c.cid AND cid = 4;
 QUERY PLAN
------------------------------------------------------------------------------------
---------------
 Nested Loop (cost=1.62..79463.79 rows=1 width=12)
 -> Subquery Scan on v (cost=1.19..79459.34 rows=1 width=8)
 Filter: (v.aid = 4)
 -> Merge Join (cost=1.19..66959.34 rows=1000000 width=8)
 Merge Cond: (a.aid = b.bid)
 -> Index Only Scan using idx_a on a (cost=0.42..25980.42 rows=1000000 width=4)
 -> Index Only Scan using idx_b on b (cost=0.42..25980.42 rows=1000000 width=4)
 -> Index Only Scan using idx_c on c (cost=0.42..4.44 rows=1 width=4)
 Index Cond: (cid = 4)
(9 rows)
```

只要看看规划师所预测的成本就知道了。成本已经从两位数飙升到惊人的数字。很明显，这个查询将为你提供糟糕的性能。

有各种方法来削弱性能，但记住优化过程是有意义的。

1.8 不断折叠

然而，在PostgreSQL中还有许多在幕后进行的优化，这些优化有助于提高整体的性能。这些功能之一被称为常量折叠。其原理是将表达式变成常量，如下面的例子所示。

```
test=# explain SELECT * FROM a WHERE aid = 3 + 1;
 QUERY PLAN
------------------------------------------------------------------------
 Index Only Scan using idx_a on a (cost=0.42..4.44 rows=1 width=4)
 Index Cond: (aid = 4)
(2 rows)
```

正如你所看到的，PostgreSQL将尝试寻找4。因为aid是有索引的，所以PostgreSQL会去进行索引扫描。请注意，我们的表只有一列，所以PostgreSQL甚至发现它需要的所有数据都可以在索引中找到。如果表达式在左边，会发生什么？

```
test=# explain SELECT * FROM a WHERE aid - 1 = 3;
 QUERY PLAN
------------------------------------------------------------------------------
 Gather (cost=1000.00..12175.00 rows=5000 width=4)
 Workers Planned: 2
 -> Parallel Seq Scan on a (cost=0.00..10675.00 rows=2083 width=4)
 Filter: ((aid - 1) = 3)
(4 rows)
test=# SET max_parallel_workers_per_gather TO 0;
SET
test=# explain SELECT * FROM a WHERE aid - 1 = 3;
 QUERY PLAN
------------------------------------------------------------
```

```
 Seq Scan on a (cost=0.00..19425.00 rows=5000 width=4)
 Filter: ((aid - 1) = 3)
(2 rows)
```

在这种情况下，索引查找代码将失败，而PostgreSQL不得不进行顺序扫描。我在这里包括两个例子--一个并行计划和一个单核计划。

为了简单起见，从现在开始，所有显示的计划都是单核心的。

1.8 理解函数内联

正如我们在本节中所概述的，有许多优化措施可以帮助加快查询速度。其中一个被称为函数内联。PostgreSQL能够内联不可变的SQL函数。其主要思想是减少必须进行的函数调用的数量，以提高速度。

下面是一个可以被优化器内联的函数的例子。

```
test=# CREATE OR REPLACE FUNCTION ld(int)
RETURNS numeric AS
$$
 SELECT log(2, $1);
$$
LANGUAGE 'sql' IMMUTABLE;
CREATE FUNCTION
```

这是一个正常的SQL函数，被标记为IMMUTABLE。这对优化器来说是完美的优化素材。简单地说，我的函数所做的就是计算一个对数。

```
test=# SELECT ld(1024);
 ld
---------------------
 10.0000000000000000
(1 row)
```

正如你所看到的，该函数按预期工作。

为了演示事情是如何进行的，我们将重新创建一个内容较少的表，以加快索引的创建过程。

```
test=# TRUNCATE a;
TRUNCATE TABLE
```

之后，可以再次添加TRUNCATE数据，并且可以应用索引。

```
test=# INSERT INTO a SELECT * FROM generate_series(1, 10000);
INSERT 0 10000
test=# CREATE INDEX idx_ld ON a (ld(aid));
CREATE INDEX
```

正如预期的那样，在该函数上创建的索引将像其他索引一样被使用。然而，让我们仔细看看索引的条件。

```
test=# EXPLAIN SELECT * FROM a WHERE ld(aid) = 10;
 QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on a (cost=4.67..52.77 rows=50 width=4)
```

```
 Recheck Cond: (log('2'::numeric, (aid)::numeric) = '10'::numeric)
 -> Bitmap Index Scan on idx_ld (cost=0.00..4.66 rows=50 width=0)
 Index Cond: (log('2'::numeric, (aid)::numeric) = '10'::numeric)
(4 rows)
test=# ANALYZE ;
ANALYZE
test=# EXPLAIN SELECT * FROM a WHERE ld(aid) = 10;
 QUERY PLAN
------------------------------------------------------------------------
 Index Scan using idx_ld on a (cost=0.29..8.30 rows=1 width=4)
 Index Cond: (log('2'::numeric, (aid)::numeric) = '10'::numeric)
(2 rows)
```

这里重要的观察是，索引条件实际上是在寻找log函数而不是ld函数。优化器已经完全摆脱了函数的调用。还值得一提的是，新鲜的优化器统计数据对于生成一个高效的计划是非常重要的。

在逻辑上，这为下面的查询打开了大门。

```
test=# EXPLAIN SELECT * FROM a WHERE log(2, aid) = 10;
 QUERY PLAN
------------------------------------------------------------------------
 Index Scan using idx_ld on a (cost=0.29..8.30 rows=1 width=4)
 Index Cond: (log('2'::numeric, (aid)::numeric) = '10'::numeric)
(2 rows)
```

优化器设法内联了这个函数，并为我们提供了一个索引扫描，这远胜于昂贵的顺序操作。

1.9 引入连接修剪

PostgreSQL也提供了一种优化，叫做连接修剪。其想法是，如果查询不需要连接，就把它们删除。如果查询是由一些中间件或ORM生成的，这就很方便了。如果一个连接可以被删除，它自然会大大加快速度，导致更少的开销。

现在的问题是，连接的修剪是如何进行的？这里有一个例子。

```
CREATE TABLE x (id int, PRIMARY KEY (id));
CREATE TABLE y (id int, PRIMARY KEY (id));
```

首先，创建两个表。确保连接条件的两边实际上都是唯一的。 这些约束条件在一分钟后将会很重要。

现在，我们可以写一个简单的查询。

```
test=# EXPLAIN SELECT *
FROM x LEFT JOIN y ON (x.id = y.id)
WHERE x.id = 3;
 QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop Left Join (cost=0.31..16.36 rows=1 width=8)
 Join Filter: (x.id = y.id)
 -> Index Only Scan using x_pkey on x
 (cost=0.15..8.17 rows=1 width=4)
 Index Cond: (id = 3)
 -> Index Only Scan using y_pkey on y
 (cost=0.15..8.17 rows=1 width=4)
```

```
 Index Cond: (id = 3)
(6 rows)
```

正如你所看到的，PostgreSQL将直接连接这些表。到目前为止，没有什么意外。然而，下面的查询被稍作修改。它不是选择所有的列，而是只选择连接的左边的那些列。

```
test=# EXPLAIN SELECT x.*
 FROM x LEFT JOIN y ON (x.id = y.id)
 WHERE x.id = 3;
 QUERY PLAN
------------------------------------------------------------------------
 Index Only Scan using x_pkey on x (cost=0.15..8.17 rows=1 width=4)
 Index Cond: (id = 3)
(2 rows)
```

PostgreSQL会直接进行内部扫描，完全跳过连接。这实际上是可能的，并且在逻辑上是正确的，有两个原因：

- 没有列从连接的右侧被选中；因此，查找这些列并不能给我们带来什么。
- 右手边是唯一的，这意味着连接不能因为右手边的重复而增加行数。

如果连接可以被自动修剪，那么查询的速度可能会快很多。这里的好处是，只需删除应用程序可能在任何情况下都不需要的列，就可以实现速度的提高。

1.10 加快集合操作

集合操作将一个以上的查询结果合并为一个结果集。 它们包括 UNION, INTERSECT, 和 EXCEPT。PostgreSQL实现了所有这些操作，并提供了许多重要的优化措施来加速它们。

规划器能够将限制推到集合操作中，为花式索引和一般的加速打开了大门。让我们看一下下面的查询，它向我们展示了这是如何工作的。

```
test=# EXPLAIN SELECT *
FROM
(
 SELECT aid AS xid
 FROM a
 UNION ALL
 SELECT bid FROM b
) AS y
WHERE xid = 3;
 QUERY PLAN
------------------------------------------------------------------------------
 Append (cost=0.29..8.76 rows=2 width=4)
 -> Index Only Scan using idx_a on a (cost=0.29..4.30 rows=1 width=4)
 Index Cond: (aid = 3)
 -> Index Only Scan using idx_b on b (cost=0.42..4.44 rows=1 width=4)
 Index Cond: (bid = 3)
(5 rows)
```

你在这里可以看到的是，两个关系被添加到对方身上。问题是，唯一的限制是在子选择之外。然而，PostgreSQL发现过滤器可以被进一步推到计划中。因此，xid = 3被附加到aid和bid上，为我们提供了在两个表上使用索引的选择。通过避免对两个表的顺序扫描，查询的运行速度会快很多。

> 注意，UNION子句和UNION ALL子句之间是有区别的。UNION ALL子句将只是盲目地追加数据，并提供两个表的结果。

```
test=# EXPLAIN SELECT *
 FROM
 (
 SELECT aid AS xid
 FROM a
 UNION SELECT bid
 FROM b
 ) AS y
WHERE xid = 3;
 QUERY PLAN
------------------------------------------------------------------------------
-----
 Unique (cost=8.79..8.80 rows=2 width=4)
 -> Sort (cost=8.79..8.79 rows=2 width=4)
 Sort Key: a.aid
 -> Append (cost=0.29..8.78 rows=2 width=4)
 -> Index Only Scan using idx_a on a (cost=0.29..4.30 rows=1 width=4)
 Index Cond: (aid = 3)
 -> Index Only Scan using idx_b on b (cost=0.42..4.44 rows=1 width=4)
 Index Cond: (bid = 3)
(8 rows)
```

执行计划看起来已经很有吸引力了。可以看到两个索引扫描。PostgreSQL必须在Append节点之上添加一个Sort节点，以确保以后可以过滤重复的内容。

> 许多没有完全意识到UNION子句和UNION ALL子句之间区别的开发者抱怨性能不好，因为他们没有意识到PostgreSQL必须过滤掉重复的数据，这在大数据集的情况下特别痛苦。

在本节中，已经讨论了一些最重要的优化。请记住，计划器内部还有很多事情要做。然而，一旦理解了最重要的步骤，编写适当的查询就更容易了。

2 了解执行计划

现在我们已经挖掘了一些在PostgreSQL中实现的重要优化，让我们继续仔细看看执行计划。你已经在本书中看到了一些执行计划。然而，为了充分利用计划，在阅读这些信息时，制定一个系统的方法是很重要的。

2.1 系统地接近计划

你必须知道的第一件事是，EXPLAIN子句可以为你做很多事情，我强烈建议充分利用这些功能。 很多人可能已经知道，EXPLAIN ANALYZE子句会执行查询并返回计划，包括真实的运行时间信息。下面是一个例子。

```
test=# EXPLAIN ANALYZE SELECT *
FROM
(
 SELECT *
 FROM b
 LIMIT 1000000
) AS b
ORDER BY cos(bid);
 QUERY PLAN
```

```
-------------------------------------------------------------------------------
---------
 Sort (cost=146173.34..148673.34 rows=1000000 width=12)
 (actual time=494.028..602.733 rows=1000000 loops=1)
 Sort Key: (cos((b.bid)::double precision))
 Sort Method: external merge Disk: 25496kB
 -> Subquery Scan on b (cost=0.00..29425.00 rows=1000000 width=12)
 (actual time=6.274..208.224 rows=1000000 loops=1)
 -> Limit (cost=0.00..14425.00 rows=1000000 width=4)
 (actual time=5.930..105.253 rows=1000000 loops=1)
 -> Seq Scan on b b_1 (cost=0.00..14425.00 rows=1000000 width=4)
 (actual time=0.014..55.448 rows=1000000 loops=1)
 Planning Time: 0.170 ms
 JIT:
 Functions: 3
 Options: Inlining false, Optimization false, Expressions true, Deforming true
 Timing: Generation 0.319 ms, Inlining 0.000 ms, Optimization 0.242 ms,
 Emission 5.196 ms, Total 5.757 ms
 Execution Time: 699.903 ms
(12 rows)
```

该计划看起来有点吓人，但不要惊慌，我们将一步一步地看下去。当阅读一个计划时，确保你从里到外阅读它。在我们的例子中，执行从对b的顺序扫描开始。这里实际上有两个信息块：成本块和实际时间块。成本块包含估算，而实际时间块是确凿的证据。它显示的是真实的执行时间。你在这里还可以看到，从PostgreSQL 12开始，JIT编译是默认开启的。这个查询已经很耗时了，足以证明JIT编译的合理性。

> 请注意，你的系统上显示的费用可能不完全相同。优化器的统计数字的微小差异会导致差异。这里最重要的是计划的读取方式。

然后，来自索引扫描的数据会被传递到Limit节点，以确保没有太多的数据。请注意，每个阶段的执行也会向我们显示所涉及的行数。 正如你所看到的，PostgreSQL首先只会从表中获取100万行；Limit节点确保这将真正发生。然而，在这个阶段有一个代价，因为运行时间已经跃升到169毫秒。最后，对数据进行排序，这需要大量的时间。在看计划时，最重要的是要弄清时间究竟损失在哪里。 做到这一点的最好方法是看一下实际的时间块，试着找出时间跳跃的地方。在这个例子中，顺序扫描需要一些时间，但它不能被大大加快。 相反，我们可以看到，随着排序的开始，时间急剧上升。

2.2 使 EXPLAIN 更冗长

在PostgreSQL中，EXPLAIN子句的输出可以加强一些，为你提供更多信息。为了尽可能多地从一个计划中提取信息，可以考虑打开以下选项。

```
test=# EXPLAIN (analyze, verbose, costs, timing, buffers)
 SELECT * FROM a ORDER BY random();
 QUERY PLAN
------------------------------------------------------------------------
 Sort (cost=834.39..859.39 rows=10000 width=12)
 (actual time=4.124..4.965 rows=10000 loops=1)
 Output: aid, (random())
 Sort Key: (random())
 Sort Method: quicksort Memory: 853kB
 Buffers: shared hit=45
 -> Seq Scan on public.a (cost=0.00..170.00 rows=10000 width=12)
 (actual time=0.057..1.457 rows=10000 loops=1)
 Output: aid, random()
 Buffers: shared hit=45
```

```
 Planning Time: 0.109 ms
 Execution Time: 5.895 ms
(10 rows)
```

analyze true将实际执行查询，如前所示。verbose true将为计划添加一些更多的信息（比如列信息）。cost true将显示关于成本的信息。timing true同样重要，因为它将为我们提供良好的运行时数据，这样我们就可以看到计划中哪里有时间被浪费。最后，还有缓冲区真值（buffers true），它可以给我们带来很大的启发。在我的例子中，它揭示了我们需要访问成千上万的缓冲区来执行查询。

在这个介绍之后，现在是时候看看其他一些主题了。如何发现计划中的问题？

2.3 发现问题

考虑到第5章 "日志文件和系统统计 "中显示的所有信息，我们已经可以发现几个潜在的性能问题，这些问题在现实生活中非常重要。

2.3.1 发现运行时的变化

当看一个计划时，你总是要问自己两个问题。

- 对于给定的查询，EXPLAIN ANALYZE子句显示的运行时间是合理的吗？
- 如果查询速度很慢，那么运行时间是在哪里跳的？

在我的例子中，连续扫描的时间是2.625毫秒。排序是在7.199毫秒后完成的，所以排序大约需要4.5毫秒来完成，因此要对查询所需的大部分运行时间负责。

寻找查询执行时间的跳跃将揭示真正发生的事情。根据哪种类型的操作会消耗过多的时间，你必须采取相应的行动。这里不可能有一些一般性的建议，因为有太多的事情会导致问题。

2.3.2 检查估算

然而，有一些事情应该一直做下去：我们应该确保估计值和实际数字合理地接近。在某些情况下，优化器会做出糟糕的决定，因为估计值由于某种原因而出现了偏差。有时，估计值会出现偏差，因为系统的统计数据没有更新。因此，运行ANALYZE子句绝对是一件好事，可以从这里开始。然而，优化器的统计资料大多是由自动真空守护进程处理，所以绝对值得考虑其他导致估算错误的选项。 看一下下面的例子，它帮助我们向一个表添加一些数据。

```
test=# CREATE TABLE t_estimate AS
 SELECT * FROM generate_series(1, 10000) AS id;
SELECT 10000
```

在加载10000行之后，优化器的统计数据被创建。

```
test=# ANALYZE t_estimate;
ANALYZE
```

让我们来看看现在的估算：

```
test=# EXPLAIN ANALYZE SELECT * FROM t_estimate WHERE cos(id) < 4;
 QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on t_estimate (cost=0.00..220.00 rows=3333 width=4)
 (actual time=0.010..4.006 rows=10000 loops=1)
 Filter: (cos((id)::double precision) < '4'::double precision)
 Planning time: 0.064 ms
 Execution time: 4.701 ms
(4 rows)
```

在很多情况下，PostgreSQL可能无法正确处理WHERE子句，因为它只有对列的统计，没有对表达式的统计。我们在这里可以看到的是对WHERE子句所返回的数据量有一个令人讨厌的低估。

当然，数据量也可能被高估，如下面的代码所示。

```
test=# EXPLAIN ANALYZE
SELECT *
FROM t_estimate
WHERE cos(id) > 4;
 QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on t_estimate (cost=0.00..220.00 rows=3333 width=4)
 (actual time=3.802..3.802 rows=0 loops=1)
 Filter: (cos((id)::double precision) > '4'::double precision)
 Rows Removed by Filter: 10000
 Planning time: 0.037 ms
 Execution time: 3.813 ms
(5 rows)
```

如果这样的事情发生在计划的深处，这个过程很可能会产生一个糟糕的计划。 因此，确保估算在一定范围内是非常合理的。

幸运的是，有一种方法可以解决这个问题。考虑一下下面的代码块。

```
test=# CREATE INDEX idx_cosine ON t_estimate (cos(id));
CREATE INDEX
```

这个列表显示了如何创建一个功能索引。

创建索引将使PostgreSQL跟踪表达式的统计数据。

```
test=# ANALYZE t_estimate;
ANALYZE
```

除了这个计划将确保显著提高性能外，它还将固定统计数据，即使索引没有被使用，如以下代码所示。

```
test=# EXPLAIN ANALYZE SELECT * FROM t_estimate WHERE cos(id) > 4;
 QUERY PLAN
-------------------------------------------------------------------
 Index Scan using idx_cosine on t_estimate
 (cost=0.29..8.30 rows=1 width=4)
 (actual time=0.002..0.002 rows=0 loops=1)
 Index Cond: (cos((id)::double precision) > '4'::double precision)
 Planning time: 0.095 ms
 Execution time: 0.011 ms
(4 rows)
```

然而，不正确的估计比看到的要多。一个经常被低估的问题被称为跨列关联。考虑一个涉及两列的简单例子。

- 20%的人喜欢滑雪。
- 20%的人来自非洲

如果我们想计算非洲的滑雪者人数，数学上说，结果将是0.2 x 0.2 = 总人口的4%。然而，非洲没有雪。因此，真正的结果肯定会更低。观察非洲和观察滑雪在统计学上并不独立。 在很多情况下，PostgreSQL保存的列统计量不跨越一个以上的列，这可能导致不好的结果。

当然，计划器做了很多工作来尽可能地防止这些事情的发生。 尽管如此，它还是会成为一个问题。

从PostgreSQL 10.0开始，我们有了多变量统计，这就一劳永逸地结束了跨列相关的问题。

2.3.3 检查缓冲区使用情况

然而，计划本身并不是唯一可能导致问题的东西。在许多情况下，危险的事情隐藏在其他一些层面上。内存和缓存会导致不希望发生的行为，对于没有经过培训的终端用户来说，往往很难理解，本节将介绍的问题就是这样。

这里有一个例子，描述了将数据随机插入到表中的情况。该查询将生成一些随机排序的数据，并将其添加到一个新表中。

```
test=# CREATE TABLE t_random AS
 SELECT * FROM generate_series(1, 10000000) AS id ORDER BY random();
SELECT 10000000
test=# ANALYZE t_random ;
ANALYZE
```

现在，我们已经生成了一个包含1000万行的简单表，并创建了优化器的统计数据。 在下一步，我们将执行一个只检索少量行的简单查询。

```
test=# EXPLAIN (analyze true, buffers true, costs true, timing true)
 SELECT * FROM t_random WHERE id < 1000;
 QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on t_random (cost=0.00..169247.71 rows=1000 width=4)
 (actual time=0.976..856.663 rows=999 loops=1)
 Filter: (id < 1000)
 Rows Removed by Filter: 9999001
 Buffers: shared hit=2080 read=42168 dirtied=14248 written=13565
 Planning Time: 0.099 ms
 Buffers: shared hit=5 dirtied=1
 Execution Time: 856.808 ms
```

在检查数据之前，要确保你已经执行了两次查询。当然，在这里使用索引是有意义的。然而，在这个查询中，PostgreSQL在缓存里面找到了2112个缓冲区，还有421136个缓冲区必须从操作系统中取出。现在，有两种情况可能发生。如果你很幸运，操作系统在缓存中找到了几个缓冲区，那么查询就很快了。如果文件系统的缓存不走运，那些块就必须从磁盘上取。这似乎是显而易见的，但是，它可能导致执行时间的巨大波动。一个完全在缓存中运行的查询可以比一个不得不慢慢从磁盘上收集随机块的查询快100倍。

让我们用一个简单的例子来概述一下这个问题。假设我们有一个存储了100亿行的电话系统（这对大型电话运营商来说并不罕见）。数据以很快的速度流入，而用户想要查询这些数据。如果你有100亿行，这些数据只能部分地装入内存，因此很多东西最终自然会来自磁盘。

让我们运行一个简单的查询来学习PostgreSQL如何查找电话号码。

```
SELECT * FROM data WHERE phone_number = '+12345678';
```

即使你在打电话，你的数据也会分散在各个地方。如果你结束一个电话只是为了开始下一个电话，成千上万的人也会这样做，所以你的两个电话最终出现在非常相同的8000个区块中的几率接近零。暂时想象一下，有10万个电话在同时进行。在磁盘上，数据将是随机分布的。如果你的电话号码经常出现，这意味着对于每一行，至少要从磁盘上获取一个块（假设有一个很低的缓存命中率）。假设有5000条记录被返回。假设你要去磁盘5,000次，这就导致了诸如5,000 x 5毫秒=25秒的执行时间。请注意，这个查询的执行时间可能在几毫秒到30秒之间，这取决于操作系统或PostgreSQL已经缓存了多少数据。

请记住，每次服务器重启都会自然而然地清除PostgreSQL和文件系统的缓存，这可能会导致节点故障后的真正麻烦。

2.3.4 修复缓冲区的高使用率

需要回答的问题是，我怎样才能改善这种情况？一种方法是运行CLUSTER子句。

```
test=# \h CLUSTER
Command: CLUSTER
Description: cluster a table according to an index
Syntax:
CLUSTER [VERBOSE] table_name [ USING index_name ]
CLUSTER [VERBOSE]
URL: https://www.postgresql.org/docs/13/sql-cluster.html
```

CLUSTER子句将以与btree索引相同的顺序重写表。如果你正在运行一个分析性工作负载，这可能是有意义的。然而，在一个OLTP系统中，CLUSTER子句可能不可行，因为在重写表的时候需要一个表锁。

修复高缓冲区使用率是很重要的。它可以带来可观的性能提升。因此，时刻关注这些问题是有意义的。

3 了解和修复连接

连接是很重要的；每个人都需要定期进行连接。因此，连接对于保持或实现良好的性能也很重要。为了确保你能写出好的连接，我们还将在本书中学习连接的知识。

3.1 获得左连接

在我们深入探讨优化连接之前，有必要看一下连接中出现的一些最常见的问题，以及其中哪些问题应该给你敲响警钟。

下面是一个简单的表结构的例子，以展示连接的工作原理。

```
test=# CREATE TABLE a (aid int);
CREATE TABLE
test=# CREATE TABLE b (bid int);
CREATE TABLE
test=# INSERT INTO a VALUES (1), (2), (3);
INSERT 0 3
test=# INSERT INTO ba VALUES (2), (3), (4);
INSERT 0 3
```

两个包含几条记录的表已经被创建。

下面的例子显示了一个简单的外层连接。

```
test=# SELECT * FROM a LEFT JOIN b ON (aid = bid);
 aid | bid
-----+-----
 1 |
 2 | 2
 3 | 3
(3 rows)
```

正如你所看到的，PostgreSQL将从左侧获取所有的记录，并且只列出符合连接条件的记录。 下面的例子可能会让很多人感到惊讶。

```
test=# SELECT * FROM a LEFT JOIN b ON (aid = bid AND bid = 2);
 aid | bid
-----+-----
 1 |
 2 | 2
 3 |
(3 rows)
```

不，行的数量不会减少--它将保持不变。大多数人认为在连接中只会有一条记录，但这并不正确，而且会导致一些隐藏的问题。

考虑一下下面的查询，它执行了一个简单的连接。

```
test=# SELECT avg(aid), avg(bid)
 FROM a LEFT JOIN b
 ON (aid = bid AND bid = 2);
 avg | avg
--------------------+--------------------
 2.0000000000000000 | 2.0000000000000000
(1 row)
```

大多数人认为，平均数是根据单行计算的。然而，正如我们前面所说，情况并非如此，因此像这样的查询通常被认为是一个性能问题，因为出于某种原因，PostgreSQL没有对连接左侧的表进行索引。当然，我们在这里看到的不是一个性能问题--我们看到的肯定是一个语义问题。通常情况下，写外连接的人并不真正知道他们在要求PostgreSQL做什么。因此，我的建议是在解决客户报告的性能问题之前，一定要质疑外联的语义正确性。

我怎么强调这种工作的重要性都不为过，因为它可以确保你的查询是正确的，并准确地完成所需的工作。

3.2 处理外连接

在验证了你的查询从商业角度来看确实是正确的之后，检查一下优化器可以做些什么来加速你的外部连接是有意义的。最重要的是，在许多情况下，PostgreSQL可以对内联接进行重新排序，从而大大地提高速度。然而，在外连接的情况下，这并不总是可能的。实际上只有少数的重排操作是允许的。

```
(A leftjoin B on (Pab)) innerjoin C on (Pac) = (A innerjoin C on (Pac)) leftjoin
B on (Pab)
```

Pac是一个指代A和C的谓词，以此类推（在这种情况下，显然，Pac不能指代B，否则转换就毫无意义了）。

- (A leftjoin B on (Pab)) leftjoin C on (Pac) = (A leftjoin C on (Pac)) leftjoin B on (Pab)
- (A leftjoin B on (Pab)) leftjoin C on (Pbc) = (A leftjoin (B leftjoin C on (Pbc)) on (Pab)

最后一条规则只有在Pbc谓词对所有空B行必须失效的情况下才成立（也就是说，Pbc对B的至少一列是严格的）。如果Pbc不严格，第一种形式可能会产生一些具有非空C列的记录，而第二种形式会使这些条目为空。

虽然有些连接可以被重新排序，但典型的查询类型不能从连接重新排序中受益。 考虑一下下面的代码片断，它有一些特殊的属性。

```
SELECT ...
 FROM a LEFT JOIN b ON (aid = bid)
 LEFT JOIN c ON (bid = cid)
 LEFT JOIN d ON (cid = did)
...
```

连接重排在这里对我们没有任何好处（因为这是不可能的）。

处理这个问题的方法是检查所有的外部连接是否真的有必要。在很多情况下，人们会在没有实际需要的情况下编写外部连接。通常情况下，商业案例甚至不需要外联。在这一节之后，我们有必要深入研究一些更多的规划器选项。

3.3 了解 join_collapse_limit 变量

在计划过程中，PostgreSQL试图检查所有可能的连接顺序。在许多情况下，这可能是相当昂贵的，因为可能有许多排列组合，这自然会减慢计划过程。

join_collapse_limit变量在这里给开发者提供了一个工具，以实际解决这些问题，并以更直接的方式定义一个查询应该如何处理。

为了告诉你这个设置是怎么回事，我们将编译一个小例子。

```
SELECT * FROM tab1, tab2, tab3
WHERE tab1.id = tab2.id
 AND tab2.ref = tab3.id;
SELECT * FROM tab1 CROSS JOIN tab2
CROSS JOIN tab3
WHERE tab1.id = tab2.id
 AND tab2.ref = tab3.id;
SELECT * FROM tab1 JOIN (tab2 JOIN tab3
 ON (tab2.ref = tab3.id))
 ON (tab1.id = tab2.id);
```

基本上，这三个查询是相同的，并且被规划器以同样的方式处理。第一个查询由隐式连接组成。最后一个只包括显式连接。在内部，计划器将检查这些请求，并相应地排列连接，以确保尽可能的最佳运行时间。这里的问题是，PostgreSQL将隐式计划多少个显式连接？这正是你可以通过设置join_collapse_limit 变量来告诉计划器的。默认值对普通查询来说是相当好的。然而，如果你的查询包含非常多的连接，使用此设置可以大大减少计划时间。减少计划时间对于保持良好的吞吐量至关重要。

为了看看join_collapse_limit变量如何改变计划，我们将编写这个简单的查询。

```
test=# EXPLAIN WITH x AS
(
 SELECT *
 FROM generate_series(1, 1000) AS id
)
SELECT *
FROM x AS a
 JOIN x AS b ON (a.id = b.id)
 JOIN x AS c ON (b.id = c.id)
 JOIN x AS d ON (c.id = d.id)
 JOIN x AS e ON (d.id = e.id)
 JOIN x AS f ON (e.id = f.id);
```

试着用不同的设置运行查询，看看计划有什么变化。不幸的是，这个计划太长了，无法在这里复制，所以不可能在这一节中包括实际的变化。

在处理了崩溃限制之后，我们现在要看一下一些额外的计划器选项。

4 启用和禁用优化器设置

到目前为止，已经详细讨论了由计划器执行的最重要的优化。多年来，PostgreSQL已经有了很大的改进。但是，还是会有一些事情发生，用户必须说服计划器做正确的事情。

为了修改计划，PostgreSQL提供了几个运行时变量，这些变量将对计划产生重大影响。这个想法是让最终用户有机会使计划中某些类型的节点比其他节点更昂贵。这在实践中意味着什么呢？下面是一个简单的计划。

```
test=# explain SELECT *
 FROM generate_series(1, 100) AS a,
 generate_series(1, 100) AS b
 WHERE a = b;
 QUERY PLAN
------------------------------------------------------------------------------------
---
 Hash Join (cost=2.25..4.63 rows=100 width=8)
 Hash Cond: (a.a = b.b)
 -> Function Scan on generate_series a (cost=0.00..1.00 rows=100 width=4)
 -> Hash (cost=1.00..1.00 rows=100 width=4)
 -> Function Scan on generate_series b (cost=0.00..1.00 rows=100 width=4)
(5 rows)
```

在这里，PostgreSQL将扫描这些函数并执行哈希连接。让我们在PostgreSQL 11或更高版本中运行同样的查询，并向你展示执行计划。

```
 QUERY PLAN
------------------------------------------------------------------------
 Merge Join (cost=119.66..199.66 rows=5000 width=8)
 Merge Cond: (a.a = b.b)
 -> Sort (cost=59.83..62.33 rows=1000 width=4)
 Sort Key: a.a
 -> Function Scan on generate_series a
 (cost=0.00..10.00 rows=1000 width=4)
 -> Sort (cost=59.83..62.33 rows=1000 width=4)
 Sort Key: b.b
 -> Function Scan on generate_series b
 (cost=0.00..10.00 rows=1000 width=4)
(8 rows)
```

你能看出这两个计划之间的区别吗？在PostgreSQL 12中，对集合返回函数的估计已经是正确的了。在旧版本中，优化器仍然估计集合返回函数将总是返回100行。在PostgreSQL中，有一些优化器支持的函数可以帮助估计结果集。因此，PostgreSQL 12及以后的计划比旧的计划有很大的优势。

在新的计划中，我们可以看到执行了一个哈希连接，当然，这是最有效的方法。然而，如果我们比优化器更聪明呢？幸运的是，PostgreSQL有办法否决优化器。你可以在连接中设置变量，改变默认的成本估算。下面是它的工作原理。

```
test=# SET enable_hashjoin TO off;
SET
test=# explain SELECT *
 FROM generate_series(1, 100) AS a,
 generate_series(1, 100) AS b
 WHERE a = b;
 QUERY PLAN
------------------------------------------------------------------------------------
---
 Merge Join (cost=8.65..10.65 rows=100 width=8)
 Merge Cond: (a.a = b.b)
 -> Sort (cost=4.32..4.57 rows=100 width=4)
 Sort Key: a.a
 -> Function Scan on generate_series a (cost=0.00..1.00 rows=100 width=4)
 -> Sort (cost=4.32..4.57 rows=100 width=4)
 Sort Key: b.b
```

```
 -> Function Scan on generate_series b (cost=0.00..1.00 rows=100 width=4)
(8 rows)
```

PostgreSQL认为hashjoin函数是坏的，并使其无限昂贵。因此，它退回到合并连接。然而，我们也可以把合并连接关闭。

```
test=# explain SELECT *
 FROM generate_series(1, 100) AS a,
 generate_series(1, 100) AS b
 WHERE a = b;
 QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop (cost=0.01..226.00 rows=100 width=8)
 Join Filter: (a.a = b.b)
 -> Function Scan on generate_series a (cost=0.00..1.00 rows=100 width=4)
 -> Function Scan on generate_series b (cost=0.00..1.00 rows=100 width=4)
(4 rows)
```

PostgreSQL正在慢慢地耗尽选项。下面的例子显示了如果我们把嵌套循环也关掉会发生什么。

```
test=# SET enable_nestloop TO off;
SET
test=# explain SELECT *
 FROM generate_series(1, 100) AS a,
 generate_series(1, 100) AS b
 WHERE a = b;
 QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop (cost=10000000000.00..10000000226.00 rows=100 width=8)
 Join Filter: (a.a = b.b)
 -> Function Scan on generate_series a (cost=0.00..1.00 rows=100 width=4)
 -> Function Scan on generate_series b (cost=0.00..1.00 rows=100 width=4)
 JIT:
 Functions: 10
 Options: Inlining true, Optimization true, Expressions true, Deforming true
(7 rows)
```

重要的是，关闭并不意味着真正的关闭--它只是意味着疯狂的昂贵。如果PostgreSQL没有更便宜的选择，它将退回到我们关闭的那些。否则，将不再有任何方法来执行SQL。

什么设置影响计划器？有以下开关。

```
# - Planner Method Configuration -
#enable_bitmapscan = on
#enable_hashagg = on
#enable_hashjoin = on
#enable_indexscan = on
#enable_indexonlyscan = on
#enable_material = on
#enable_mergejoin = on
#enable_nestloop = on
#enable_parallel_append = on
#enable_seqscan = on
#enable_sort = on
```

```
#enable_incrementalsort = on
#enable_tidscan = on
#enable_partitionwise_join = off
#enable_partitionwise_aggregate = off
#enable_parallel_hash = on
#enable_partition_pruning = on
```

虽然这些设置肯定是有益的，但请理解这些调整应该被谨慎处理。它们应该只用于加快个别查询，而不是全盘关闭。关掉选项可以很快对你不利，并破坏性能。 因此，在改变这些参数之前，真的要三思而后行。

然而，穷举式搜索并不是优化查询的唯一方法。还有一个遗传查询优化器，这将在下一节介绍。

4.1 了解遗传查询优化

规划过程的结果是实现卓越绩效的关键。正如我们在本章中所看到的，规划远非简单明了，它涉及各种复杂的计算。一个查询所涉及的表越多，规划就会变得越复杂。表越多，规划者的选择就越多。从逻辑上讲，规划的时间会增加。在某些时候，计划会花费如此长的时间，以至于执行经典的穷举搜索不再可行。 除此之外，在规划过程中发生的错误是如此之大，以至于找到理论上最好的计划不一定能在运行时间上导致最好的计划。

在这种情况下，遗传查询优化（GEQO）可以起到救急的作用。什么是GEQO？ 这个想法来自于自然界的灵感，类似于自然界的进化过程。

PostgreSQL会像处理旅行推销员问题一样处理这个问题，并将可能的连接编码为整数串。例如，4-1-3-2意味着先连接4和1，然后是3，然后是2。 这些数字代表关系的ID。

首先，遗传优化器将生成一组随机的计划。然后对这些计划进行检查。坏的计划被丢弃，新的计划则根据好的计划的基因生成。这样一来，就有可能产生更好的计划。这个过程可以根据需要经常重复。在一天结束时，我们留下的计划预计会比仅仅使用随机计划要好得多。 GEQO可以通过调整geqo变量来开启和关闭，如下面几行代码所示。

```
test=# SHOW geqo;
 geqo
------
 on
(1 row)
test=# SET geqo TO off;
SET
```

该清单显示了GEQO如何被打开和关闭。默认情况下，如果一个语句超过一定的复杂程度，geqo变量就会启动，这由下面的变量控制。

```
test=# SHOW geqo_threshold ;
 geqo_threshold
----------------
 12
(1 row)
```

如果你的查询量太大，以至于你开始达到这个阈值，规划器如何更改计划当然是有意义的，看看如果你改变这些变量，计划器会如何改变计划。

然而，作为一般规则，我建议尽量避免使用GEQO，并尝试首先通过使用join_collapse_limit变量来修复连接顺序。请注意，每个查询都是不同的，所以通过学习计划器在不同情况下的表现，进行实验并获得更多的经验肯定会有帮助。

> 如果你想了解更多关于连接的信息，请参考以下链接：http://de.slideshare.net/hansjurgenschonig/postgresql-joining-1-million-tables。

在下一节中，我们将看一下分区，这是一种将大数据集分割成小块的方法。

5 分区数据

鉴于默认的8,000个块，PostgreSQL可以在一个表中存储多达32TB的数据。如果你用32,000个块来编译PostgreSQL，你甚至可以把多达128TB的数据放到一个表中。 然而，像这样的大表不一定很方便了，对表进行分区可以使处理更容易，在某些情况下还可以更快一点。从10.0版本开始，PostgreSQL提供了改进的分区，这将为终端用户提供明显的处理数据分区的方便。

在本章中，将介绍旧的分区手段，以及从PostgreSQL 13.0开始提供的新功能。在我们说话的时候，分区方面的功能在各个领域都在增加，因此人们可以期待在PostgreSQL的所有未来版本中进行更多更好的分区。你将学习的第一件事是如何使用经典的PostgreSQL继承。

5.1 创建继承表

首先，我们将仔细研究一下过时的数据分区方法。请记住，了解这种技术对于深入了解PostgreSQL在幕后的真正作用非常重要。

在深入挖掘分区的优势之前，我想向你展示如何创建分区。整个事情从一个父表开始，我们可以用下面的命令来创建。

```
test=# CREATE TABLE t_data (id serial, t date, payload text);
CREATE TABLE
```

在这个例子中，父表有三列。日期列将被用于分区，但我们稍后将详细介绍。

现在，父表已经到位，可以创建子表了。它是这样工作的。

```
test=# CREATE TABLE t_data_2016 () INHERITS (t_data);
CREATE TABLE
test=# \d t_data_2016
 Table "public.t_data_2016"
 Column | Type | Collation | Nullable | Default
--------+---------+-----------+----------+-----------------------------------
 id | integer | | not null | nextval('t_data_id_seq'::regclass)
 t | date | | |
 payload | text | | |
Inherits: t_data
```

该表被称为t_data_2016，继承自t_data。(). 这意味着没有额外的列被添加到子表中。正如你所看到的，继承意味着所有来自父表的列都可以在子表中使用。还要注意的是，id列将继承父表的序列，这样所有的子表就可以共享非常相同的编号。 让我们再创建一些表。

```
test=# CREATE TABLE t_data_2015 () INHERITS (t_data);
CREATE TABLE
test=# CREATE TABLE t_data_2014 () INHERITS (t_data);
CREATE TABLE
```

到目前为止，所有的表都是相同的，只是继承了父表。然而，还有一点：子表实际上可以比父表有更多的列。添加更多的字段很简单。

```
test=# CREATE TABLE t_data_2013 (special text) INHERITS (t_data);
CREATE TABLE
```

在这种情况下，已经添加了一个特殊的列。它对父表没有影响；它只是丰富了子表，使它们能够容纳更多的数据。 在创建了少量的表之后，可以添加一行。

```
test=# INSERT INTO t_data_2015 (t, payload)
 VALUES ('2015-05-04', 'some data');
INSERT 0 1
```

现在最重要的是，父表可以用来寻找子表中的所有数据。

```
test=# SELECT * FROM t_data;
 id | t | payload
----+------------+-----------
 1 | 2015-05-04 | some data
(1 row)
```

查询父类允许你以简单有效的方式访问父类下面的一切。

为了理解PostgreSQL是如何进行分区的，看一下计划是有意义的。

```
test=# EXPLAIN SELECT * FROM t_data;
 QUERY PLAN
-------------------------------------------------------------------------------
 Append (cost=0.00..106.16 rows=4411 width=40)
 -> Seq Scan on t_data t_data_1 (cost=0.00..0.00 rows=1 width=40)
 -> Seq Scan on t_data_2016 t_data_2 (cost=0.00..22.00 rows=1200 width=40)
 -> Seq Scan on t_data_2015 t_data_3 (cost=0.00..22.00 rows=1200 width=40)
 -> Seq Scan on t_data_2014 t_data_4 (cost=0.00..22.00 rows=1200 width=40)
 -> Seq Scan on t_data_2013 t_data_5 (cost=0.00..18.10 rows=810 width=40)
(6 rows)
```

实际上，这个过程是非常简单的。PostgreSQL将简单地统一所有的表，并向我们显示我们正在看的分区内部和下面的所有表的所有内容。请注意，所有的表都是独立的，只是通过系统目录进行逻辑连接，但如果有一种方法可以使优化器的决策更加智能呢?

5.2 应用表约束

如果对该表应用过滤器会发生什么？为了以最有效的方式执行这个查询，优化器会决定怎么做？下面的例子向我们展示了PostgreSQL规划器的行为方式。

```
test=# EXPLAIN SELECT * FROM t_data WHERE t = '2016-01-04';
 QUERY PLAN
-------------------------------------------------------------------------------
```

```
 Append (cost=0.00..95.24 rows=23 width=40)
 -> Seq Scan on t_data t_data_1 (cost=0.00..0.00 rows=1 width=40)
 Filter: (t = '2016-01-04'::date)
 -> Seq Scan on t_data_2016 t_data_2 (cost=0.00..25.00 rows=6 width=40)
 Filter: (t = '2016-01-04'::date)
 -> Seq Scan on t_data_2015 t_data_3 (cost=0.00..25.00 rows=6 width=40)
 Filter: (t = '2016-01-04'::date)
 -> Seq Scan on t_data_2014 t_data_4 (cost=0.00..25.00 rows=6 width=40)
 Filter: (t = '2016-01-04'::date)
 -> Seq Scan on t_data_2013 t_data_5 (cost=0.00..20.12 rows=4 width=40)
 Filter: (t = '2016-01-04'::date)
(11 rows)
```

PostgreSQL将把这个过滤器应用到结构中的所有分区。它不知道表名与表的内容有某种联系。对数据库来说，名字只是名字，与我们要找的东西没有关系。当然，这是有道理的，因为没有任何数学上的理由来做其他事情。

现在的问题是：我们怎样才能告诉数据库，2016年的表只包含2016年的数据，2015年的表只包含2015年的数据，以此类推？表约束在这里正是为了做这个。它们教导PostgreSQL关于这些表的内容，因此允许规划器做出比以前更聪明的决定。这个功能被称为约束条件排除，在许多情况下有助于大幅提高查询速度。

下面的列表显示了如何创建表约束。

```
test=# ALTER TABLE t_data_2013
 ADD CHECK (t < '2014-01-01');
ALTER TABLE
test=# ALTER TABLE t_data_2014
 ADD CHECK (t >= '2014-01-01' AND t < '2015-01-01');
ALTER TABLE
test=# ALTER TABLE t_data_2015
 ADD CHECK (t >= '2015-01-01' AND t < '2016-01-01');
ALTER TABLE
test=# ALTER TABLE t_data_2016
 ADD CHECK (t >= '2016-01-01' AND t < '2017-01-01');
ALTER TABLE
```

对于每个表，可以添加一个 CHECK 约束。

> PostgreSQL只有在这些表中的所有数据都完全正确，并且每一行都满足约束条件的情况下才会创建约束条件。与MySQL相比，PostgreSQL中的约束被认真对待，在任何情况下都会被尊重。

在PostgreSQL中，这些约束可以重叠--这并不被禁止，在某些情况下是有意义的。然而，通常情况下，不重叠的约束是更好的，因为PostgreSQL可以选择修剪更多的表。

下面是添加这些表约束后的情况。

```
test=# EXPLAIN SELECT * FROM t_data WHERE t = '2016-01-04';
 QUERY PLAN
------------------------------------------------------------------------------
 Append (cost=0.00..25.04 rows=7 width=40)
 -> Seq Scan on t_data t_data_1 (cost=0.00..0.00 rows=1 width=40)
 Filter: (t = '2016-01-04'::date)
 -> Seq Scan on t_data_2016 t_data_2 (cost=0.00..25.00 rows=6 width=40)
 Filter: (t = '2016-01-04'::date)
(5 rows)
```

计划员将能够从查询中删除许多表，只保留那些可能包含数据的表。这个查询可以从一个更短、更有效的计划中大大受益。特别是，如果这些表真的很大，删除它们可以大大地提高速度。

在下一步，你将学习如何修改这些结构。

5.3 修改继承结构

偶尔，数据结构必须被修改。ALTER TABLE子句在这里正是为了做这个。这里的问题是，如何修改分区表?

基本上，你所要做的就是处理父表并添加或删除列。PostgreSQL会自动将这些修改传播到子表，并确保对所有关系进行修改，如下所示。

```
test=# ALTER TABLE t_data ADD COLUMN x int;
ALTER TABLE
test=# \d t_data_2016
 Table "public.t_data_2016"
 Column | Type | Collation | Nullable | Default
---------+---------+-----------+----------+-------------------------------------
 id | integer | | not null | nextval('t_data_id_seq'::regclass)
 t | date | | |
 payload | text | | |
 x | integer | | |
Check constraints:
 "t_data_2016_t_check" CHECK (t >= '2016-01-01'::date AND t < '2017-01-
01'::date)
Inherits: t_data
```

正如你所看到的，该列被添加到父表并自动添加到子表这里。

请注意，这对列也是有效的。索引是一个完全不同的故事。在一个继承结构中，每个表都必须单独建立索引。如果你向父表添加索引，它将只存在于父表上--它不会被部署在那些子表上。为所有这些表中的所有这些列建立索引是你的任务，PostgreSQL不会为你做这些决定。当然，这可以被看作是一种功能，也可以被看作是一种限制。从好的方面看，你可以说PostgreSQL给了你单独索引事物的灵活性，因此有可能更有效率。 然而，人们也可以说，一个一个地部署所有这些索引是一个更多的工作。

5.4 将表移入和移出分区结构

假设你有一个继承的结构。数据是按日期划分的，你想向终端用户提供最近几年的数据。在某些时候，你可能想从用户的范围内删除一些数据，而不实际接触它。你可能想把数据放到某种存档中。

PostgreSQL提供了一个简单的方法来实现这个目的。首先，可以创建一个新的父类。

```
test=# CREATE TABLE t_history (LIKE t_data);
CREATE TABLE
```

LIKE关键字允许你创建一个与t_data表布局完全相同的表。如果你忘记了t_data表到底有哪些列，这可能会派上用场，因为它为你节省了很多工作。它还可以包括索引、约束和默认值。

然后，该表可以从旧的父表上移开，放在新表的下面。下面是它的工作原理。

```
test=# ALTER TABLE t_data_2013 NO INHERIT t_data;
ALTER TABLE
test=# ALTER TABLE t_data_2013 INHERIT t_history;
ALTER TABLE
```

当然，整个过程可以在一个事务中完成，以确保操作保持原子性。

5.5 清理数据

分区表的一个优点是能够快速清理数据。让我们假设我们想删除一整年的数据。如果数据被相应地分区，一个简单的DROP TABLE子句就可以完成这个工作。

```
test=# DROP TABLE t_data_2014;
DROP TABLE
```

正如你所看到的，删除一个子表很容易。但是父表呢？有依赖对象，所以PostgreSQL自然会出错，以确保没有意外发生。

```
test=# DROP TABLE t_data;
ERROR: cannot drop table t_data because other objects depend on it
DETAIL: default for table t_data_2013 column id depends on
 sequence t_data_id_seq
table t_data_2016 depends on table t_data
table t_data_2015 depends on table t_data
HINT: Use DROP ... CASCADE to drop the dependent objects too.
```

DROP TABLE子句会警告我们存在依赖对象，并拒绝删除这些表。下面的例子向我们展示了如何使用一个级联的DROP TABLE。

```
test=# DROP TABLE t_data CASCADE;
NOTICE: drop cascades to 3 other objects
DETAIL: drop cascades to default value for column id of table t_data_2013
drop cascades to table t_data_2016
drop cascades to table t_data_2015
DROP TABLE
```

需要使用CASCADE子句来强迫PostgreSQL实际删除这些对象，以及父表。 在介绍了经典的手段之后，我们现在将看看高级的PostgreSQL 13.x分区。

5.6 了解PostgreSQL 13.x的分区功能

自从分区引入后，PostgreSQL增加了很多东西，很多你在旧世界看到的东西从那时起都被自动化或变得更容易。然而，让我们以一种更有条理的方式来看看这些东西。

许多年来，PostgreSQL社区一直在研究内置分区。最后，PostgreSQL 10.0提供了第一个内核分区的实现。在PostgreSQL 10中，分区功能仍然相当基本，因此在PostgreSQL 11、12和现在的13中改进了很多东西，使想使用这一重要功能的人的生活更加容易。

为了向你展示分区的工作原理，我编译了一个以范围分区为特征的简单例子，如下所示。

```
CREATE TABLE data (
 payload integer
) PARTITION BY RANGE (payload);
CREATE TABLE negatives PARTITION
 OF data FOR VALUES FROM (MINVALUE) TO (0);

CREATE TABLE positives PARTITION
 OF data FOR VALUES FROM (0) TO (MAXVALUE);
```

在这个例子中，一个分区将保存所有的负值，而另一个分区将处理正值。在创建父表时，你可以简单地指定你想要的数据分区的方式。 一旦父表被创建，现在是创建分区的时候了。要做到这一点，必须添加PARTITION OF子句。在PostgreSQL 10中，仍然有一些限制。最重要的是，一个元组（行）不能从一个分区移动到另一个分区，如下所示。

```
UPDATE data SET payload = -10 WHERE payload = 5
```

幸运的是，这个限制已经被取消了，PostgreSQL 11能够将一行从一个分区移动到另一个分区。然而，请记住，在分区之间移动数据可能不是一般的好主意。

让我们来看看幕后发生了什么：

```
test=# INSERT INTO data VALUES (5);
INSERT 0 1
test=# SELECT * FROM data;
 payload
---------
 5
(1 row)
test=# SELECT * FROM positives;
 payload
---------
 5
(1 row)
```

数据被移到了正确的分区中。如果我们改变这个值，你会看到分区也会改变。下面的列表显示了这方面的一个例子。

```
test=# UPDATE data
 SET payload = -10
 WHERE payload = 5
 RETURNING *;
 payload
---------
 -10
(1 row)
UPDATE 1
test=# SELECT * FROM negatives;
 payload
---------
 -10
(1 row)
```

每一行都被放到了正确的表中，如前面的列表所示。 下一个重要的方面是与索引有关。在PostgreSQL 10中，每个表（每个分区）都必须单独编制索引。在PostgreSQL 11及更高版本中，这不再是事实。让我们来试试这个，看看会发生什么。

```
test=# CREATE INDEX idx_payload ON data (payload);
CREATE INDEX
test=# \d positives
 Table "public.positives"
 Column | Type | Collation | Nullable | Default
---------+---------+-----------+----------+---------
 payload | integer | | |
Partition of: data FOR VALUES FROM (0) TO (MAXVALUE)
Indexes:
 "positives_payload_idx" btree (payload)
```

你在这里可以看到的是，索引也被自动添加到子表中，这是PostgreSQL 11的一个非常重要的特性，并且已经被将他们的应用程序转移到PostgreSQL 11及以后的用户广泛赞赏。

另一个重要的功能是能够创建一个默认的分区。为了向你展示它是如何工作的，我们可以放弃我们两个分区中的一个。

```
test=# DROP TABLE negatives;
DROP TABLE
```

然后，可以轻松创建数据表的默认分区：

```
test=# CREATE TABLE p_def PARTITION OF data DEFAULT;
CREATE TABLE
```

所有不适合的数据都会在这个默认分区中结束，这确保了创建正确的分区永远不会被遗忘。经验表明，随着时间的推移，默认分区的存在使应用程序更加可靠。

在本节中，你已经了解了分区的知识。在下一节中，我们将指导你学习一些更高级的性能参数

6 调整参数以获得良好的查询性能

编写好的查询是实现良好性能的第一步。没有一个好的查询，你很可能会遭受糟糕的性能。因此，编写好的、智能的代码会给你带来最大的优势。一旦你的查询从逻辑和语义的角度进行了优化，良好的内存设置可以为你提供一个不错的最终加速。

在本节中，我们将学习更多的内存可以为你做什么，以及PostgreSQL如何使用它为你带来好处。同样，本节假设我们使用单核查询，使计划更易读。为了确保始终只有一个核心在工作，使用下面的命令。

```
test=# SET max_parallel_workers_per_gather TO 0;
SET
```

下面是一个简单的例子，演示内存参数可以为你做什么。

```
test=# CREATE TABLE t_test (id serial, name text);
CREATE TABLE
test=# INSERT INTO t_test (name)
 SELECT 'hans' FROM generate_series(1, 100000);
INSERT 0 100000
test=# INSERT INTO t_test (name)
 SELECT 'paul' FROM generate_series(1, 100000);
INSERT 0 100000
```

100万条包含hans的记录将被添加到表中。然后，100万条包含paul的记录将被加载。总的来说，将有200万个唯一的ID，但只有两个不同的名字。

让我们通过使用PostgreSQL的默认内存设置运行一个简单的查询。

```
test=# SELECT name, count(*) FROM t_test GROUP BY 1;
 name | count
-------+--------
 hans | 100000
 paul | 100000
(2 rows)
```

将会返回两行，这不足为奇。这里重要的不是结果，而是PostgreSQL在幕后做了什么。

```
test=# EXPLAIN ANALYZE SELECT name, count(*)
 FROM t_test
 GROUP BY 1;
 QUERY PLAN
------------------------------------------------------------------------------
 HashAggregate (cost=4082.00..4082.01 rows=1 width=13)
 (actual time=59.876..59.877 rows=2 loops=1)
 Group Key: name
 Peak Memory Usage: 24 kB
 -> Seq Scan on t_test (cost=0.00..3082.00 rows=200000 width=5)
 (actual time=0.009..24.186 rows=200000 loops=1)
 Planning Time: 0.052 ms
 Execution Time: 59.907 ms
(6 rows)
```

PostgreSQL发现，组的数量实际上是非常小的。因此，它创建了一个哈希，为每个组添加一个哈希条目，然后开始计数。由于组的数量少，哈希值真的很小，PostgreSQL可以通过增加每个组的数字来快速进行计数。

如果我们按ID而不是按名字分组会发生什么？组的数量会激增。在PostgreSQL 13中，已经实现了一项改进：哈希值现在可以溢出到磁盘。

```
test=# EXPLAIN ANALYZE SELECT id, count(*) FROM t_test GROUP BY 1;
 QUERY PLAN
------------------------------------------------------------------------------
 HashAggregate (cost=7207.00..9988.25 rows=200000 width=12)
 (actual time=76.609..140.297 rows=200000 loops=1)
 Group Key: id
 Planned Partitions: 8 Peak Memory Usage: 4177 kB
 Disk Usage: 7680 kB
 HashAgg Batches: 8
 -> Seq Scan on t_test (cost=0.00..3082.00 rows=200000 width=4)
 (actual time=0.008..22.947 rows=200000 loops=1)
 Planning Time: 0.115 ms
 Execution Time: 270.249 ms
(6 rows)
```

前面列表中的执行计划给了我们一些很好的启示。它显示了哪些操作是需要的。

在PostgreSQL中，后备策略是使用一个GroupAggregate。你可以很容易地模拟以前的行为。

```
test=# SET enable_hashagg TO off;
SET
```

替代计划显示在以下代码片段中：

```
test=# EXPLAIN ANALYZE SELECT id, count(*) FROM t_test GROUP BY 1;
 QUERY PLAN
------------------------------------------------------------------------------------
 GroupAggregate (cost=23428.64..26928.64 rows=200000 width=12)
 (actual time=55.259..130.352 rows=200000 loops=1)
 Group Key: id
 -> Sort (cost=23428.64..23928.64 rows=200000 width=4)
 (actual time=55.250..75.275 rows=200000 loops=1)
 Sort Key: id
 Sort Method: external merge Disk: 3328kB
 -> Seq Scan on t_test (cost=0.00..3082.00 rows=200000 width=4)
 (actual time=0.009..21.601 rows=200000 loops=1)
 Planning Time: 0.046 ms
 Execution Time: 153.923 ms
(8 rows)
```

PostgreSQL发现现在组的数量大了很多，于是迅速改变策略。问题是，一个包含这么多条目的哈希值不适合放在内存中。

```
test=# SHOW work_mem ;
 work_mem
----------
 4MB
(1 row)
```

我们可以看到，work_mem变量控制着GROUP BY子句所使用的哈希的大小。 由于条目太多，PostgreSQL必须找到一种策略，不要求我们在内存中保存整个数据集。解决方案是按ID对数据进行排序并分组。一旦数据被排序，PostgreSQL就可以向下移动列表，形成一个又一个组。如果第一种类型的值被计算出来，部分结果就会被读取并可以被排放出来。然后，可以处理下一个组。一旦排序列表中的值在向下移动时发生变化，它就不会再出现；因此，系统知道部分结果已经准备就绪。

为了加快查询速度，可以在运行中为work_mem变量设置一个较高的值（当然，也可以是全局的）。

```
test=# SET work_mem TO '1 GB';
SET
```

现在，该计划将再次以快速和高效的哈希聚合为特征。

```
test=# EXPLAIN ANALYZE SELECT id, count(*) FROM t_test GROUP BY 1;
 QUERY PLAN
-------------------------------------------------------------------
 HashAggregate (cost=4082.00..6082.00 rows=200000 width=12)
 (actual time=76.967..118.926 rows=200000 loops=1)
 Group Key: id
 -> Seq Scan on t_test
 (cost=0.00..3082.00 rows=200000 width=4)
 (actual time=0.008..13.570 rows=200000 loops=1)
 Planning time: 0.073 ms
 Execution time: 126.456 ms
(5 rows)
```

PostgreSQL知道（或至少假设）数据将适合于内存并切换到更快的计划。 正如你所看到的，执行时间更短。查询不会像GROUP BY名字的情况那样快，因为要计算更多的哈希值，但是在绝大多数情况下，你将能够看到一个很好的、可靠的好处。如前所述，这种行为有点依赖于版本。

6.1 加快排序

work_mem变量不仅可以加快分组的速度。它还可以对简单的事情产生非常好的影响，比如排序，这是一个被世界上每个数据库系统掌握的基本机制。

下面的查询显示了一个使用默认设置为4MB的简单操作。

```
test=# SET work_mem TO default;
SET
test=# EXPLAIN ANALYZE SELECT * FROM t_test ORDER BY name, id;
 QUERY PLAN
------------------------------------------------------------------------------
 Sort (cost=24111.14..24611.14 rows=200000 width=9)
 (actual time=60.338..89.218 rows=200000 loops=1)
 Sort Key: name, id
 Sort Method: external merge Disk: 6912kB
 -> Seq Scan on t_test (cost=0.00..3082.00 rows=200000 width=9)
 (actual time=0.006..17.818 rows=200000 loops=1)
 Planning Time: 0.074 ms
 Execution Time: 162.544 ms
(6 rows)
```

PostgreSQL需要17.8毫秒来读取数据，超过70毫秒来排序。由于可用的内存量很低，排序必须使用临时文件进行。外部合并磁盘方法只需要少量的RAM，但必须将中间数据发送到一个相对较慢的存储设备上，这当然会导致吞吐量不佳。

增加work_mem变量的设置将使PostgreSQL使用更多的内存进行排序。

```
test=# SET work_mem TO '1 GB';
SET
test=# EXPLAIN ANALYZE SELECT * FROM t_test ORDER BY name, id;
 QUERY PLAN
-------------------------------------------------------------------
 Sort (cost=20691.64..21191.64 rows=200000 width=9)
 (actual time=36.481..47.899 rows=200000 loops=1)
 Sort Key: name, id
 Sort Method: quicksort Memory: 15520kB
 -> Seq Scan on t_test
```

```
 (cost=0.00..3082.00 rows=200000 width=9)
 (actual time=0.010..14.232 rows=200000 loops=1)
 Planning time: 0.037 ms
 Execution time: 55.520 ms
(6 rows)
```

由于现在有足够的内存，数据库将在内存中完成所有的排序，因此大大加快了这个过程。现在的排序只需要33毫秒，与我们之前的查询相比，有7倍的改进。更多的内存将导致更快的排序，并将加快系统的速度。

到目前为止，你已经看到了两种可用于排序数据的机制：外部合并磁盘和快速排序内存。除了这两种机制之外，还有第三种算法，那就是top-N heapsort Memory。它可以用来只为你提供前N行的数据。

```
test=# EXPLAIN ANALYZE SELECT * FROM t_test ORDER BY name, id LIMIT 10;
 QUERY PLAN
------------------------------------------------------------------------
 Limit (cost=7403.93..7403.95 rows=10 width=9)
 (actual time=31.837..31.838 rows=10 loops=1)
 -> Sort (cost=7403.93..7903.93 rows=200000 width=9)
 (actual time=31.836..31.837 rows=10 loops=1)
 Sort Key: name, id
 Sort Method: top-N heapsort Memory: 25kB
 -> Seq Scan on t_test
 (cost=0.00..3082.00 rows=200000 width=9)
 (actual time=0.011..13.645 rows=200000 loops=1)
 Planning time: 0.053 ms
 Execution time: 31.856 ms
(7 rows)
```

该算法快如闪电，整个查询将在30多毫秒内完成。排序部分现在只需要18毫秒，因此几乎和首先读取数据一样快。

在PostgreSQL 13中，增加了一种新的算法。

```
test=# CREATE INDEX idx_id ON t_test (id);
CREATE INDEX
test=# explain analyze SELECT * FROM t_test ORDER BY id, name;
 QUERY PLAN
------------------------------------------------------------------------------------------
---
 Incremental Sort (cost=0.46..15289.42 rows=200000 width=9)
 (actual time=0.047..71.622 rows=200000 loops=1)
 Sort Key: id, name
 Presorted Key: id
 Full-sort Groups: 6250 Sort Method: quicksort
 Average Memory: 26kB Peak Memory: 26kB
 -> Index Scan using idx_id on t_test (cost=0.42..6289.42 rows=200000 width=9)
 (actual time=0.032..37.965 rows=200000 loops=1)
 Planning Time: 0.165 ms
 Execution Time: 83.681 ms
(7 rows)
```

如果数据已经被某些变量排序，则使用增量排序。在这种情况下，idx_id将返回按id排序的数据。我们所要做的就是对已经排序的数据按名称进行排序。

注意，work_mem变量是按操作分配的。理论上，一个查询可能需要work_mem变量不止一次。这不是一个全局设置--它确实是按操作分配的。因此，你必须以一种谨慎的方式来设置它。

我们需要记住的一点是，有许多书声称，在OLTP系统上将work_mem变量设置得过高，可能会导致你的服务器内存耗尽。是的；如果1,000人同时分拣100MB，这可能导致内存失效。然而，你期望磁盘能够处理这种情况吗？我很怀疑。解决方案只能是重新思考你正在做的事情。无论如何，在一个OLTP系统中，对100MB的数据进行1000次并发排序是不应该发生的。考虑部署适当的索引，编写更好的查询，或者干脆重新思考你的要求。在任何情况下，如此频繁地并发排序如此多的数据都是一个坏主意--在这些事情停止你的应用之前停止。

6.2 加快管理任务

有更多的操作实际上需要做一些排序或某种内存分配。 像CREATE INDEX子句这样的管理型操作不依赖于work_mem变量，而是使用maintenance_work_mem变量。下面是它的工作原理。

```
test=# DROP INDEX idx_id;
DROP INDEX
test=# SET maintenance_work_mem TO '1 MB';
SET
test=# \timing
Timing is on.
test=# CREATE INDEX idx_id ON t_test (id);
CREATE INDEX
Time: 104.268 ms
```

正如你所看到的，在200万行上创建一个索引需要大约100毫秒，这确实很慢。因此，maintenance_work_mem变量可以用来加快排序速度，这基本上就是CREATE INDEX子句的作用。

```
test=# SET maintenance_work_mem TO '1 GB';
SET
test=# CREATE INDEX idx_id2 ON t_test (id);
CREATE INDEX
Time: 46.774 ms
```

现在的速度已经翻了一番，就因为排序已经得到了很大的改善。

还有更多的管理作业可以从更多的内存中受益。最突出的是VACUUM子句（用于清理索引）和ALTER TABLE子句。maintenance_work_mem变量的规则和work_mem变量的规则是一样的。设置是按操作进行的，只有所需的内存才会被即时分配。

在PostgreSQL 11中，数据库引擎增加了一个额外的功能。PostgreSQL现在能够并行地建立btree索引，这这可以显着加快大表的索引速度。负责配置并行性的参数如下。

```
test=# SHOW max_parallel_maintenance_workers;
 max_parallel_maintenance_workers
----------------------------------
 2
(1 row)
```

max_parallel_maintenance_workers 控制CREATE INDEX可以使用的最大工作进程数量。至于每一个并行操作，PostgreSQL会根据表的大小来决定worker的数量。当为大表建立索引时，索引创建可以看到大幅度的改进。我做了一些广泛的测试，并在我的一篇博文中总结了我的发现：https://www.cybertec-postgresql.com/en/postgresql-parallel-create-index-for-better-performance/。在这里，你将了解到与索引创建有关的一些重要的性能洞察力。 然而，索引创建并不是唯一支持并发性的东西。

7 利用并行查询

从9.6版本开始，PostgreSQL支持并行查询。随着时间的推移，这种对并行的支持逐渐得到改善，11版为这一重要功能增加了更多的功能。在这一节中，我们将看看并行是如何工作的，以及可以做些什么来加速工作。

在深入了解这些细节之前，有必要创建一些样本数据，如下所示。

```
test=# CREATE TABLE t_parallel AS
 SELECT * FROM generate_series(1, 25000000) AS id;
SELECT 25000000
```

在加载初始数据后，我们可以运行我们的第一个并行查询。一个简单的计数将显示出并行查询的一般情况。

```
test=# explain SELECT count(*) FROM t_parallel;
 QUERY PLAN
------------------------------------------------------------------------------------------
----------------
 Finalize Aggregate (cost=241829.17..241829.18 rows=1 width=8)
 -> Gather (cost=241828.96..241829.17 rows=2 width=8)
 Workers Planned: 2
 -> Partial Aggregate (cost=240828.96..240828.97 rows=1 width=8)
 -> Parallel Seq Scan on t_parallel (cost=0.00..214787.17 rows=10416717 width=0)
(5 rows)
```

让我们详细看看这个查询的执行计划。首先，PostgreSQL执行了一个并行的顺序扫描。这意味着PostgreSQL将使用1个以上的CPU来处理这个表（逐块处理），它将创建部分聚合。收集节点的工作是收集数据，并将其传递给做最后的聚合。聚集节点是并行性的终点。重要的是要提到并行性（目前）从不嵌套。在另一个聚集节点中不可能有一个聚集节点。在这个例子中，PostgreSQL决定采用两个工作进程。这是为什么呢？ 让我们考虑一下下面这个变量。

```
test=# SHOW max_parallel_workers_per_gather;
 max_parallel_workers_per_gather
---------------------------------
 2
(1 row)
```

max_parallel_workers_per_gather限制了聚集节点下面允许的工作进程的数量为两个。重要的是：如果一个表很小，它将永远不会使用并行性。一个表的大小必须至少是8MB，由以下配置设置定义。

```
test=# SHOW min_parallel_table_scan_size;
 min_parallel_table_scan_size
------------------------------
 8MB
(1 row)
```

现在，并行的规则如下：表的大小必须是三倍，才能让PostgreSQL增加一个工作进程。换句话说，要想获得四个额外的工作者，你至少需要81倍的数据。这是有道理的，因为你的数据库的大小上升了100倍，而存储系统的速度通常不是100倍。因此，有用的核心数量是有限的。

然而，我们的表是相当大的。

```
test=# \d+
 List of relations
 Schema | Name | Type | Owner | Persistence | Size | Description
--------+------------+-------+-------+-------------+--------+-------------
 public | t_parallel | table | hs | permanent | 864 MB |
(1 row)
```

在这个例子中，max_parallel_workers_per_gather限制了内核的数量。如果我们改变这个设置，PostgreSQL将决定更多的核心。

```
test=# SET max_parallel_workers_per_gather TO 10;
SET
test=# explain SELECT count(*) FROM t_parallel;
 QUERY PLAN
------------------------------------------------------------------------------------------
---------------
 Finalize Aggregate (cost=174120.82..174120.83 rows=1 width=8)
 -> Gather (cost=174120.30..174120.81 rows=5 width=8)
 Workers Planned: 5
 -> Partial Aggregate (cost=173120.30..173120.31 rows=1 width=8)
 -> Parallel Seq Scan on t_parallel (cost=0.00..160620.24 rows=5000024 width=0)
 JIT:
 Functions: 4
 Options: Inlining false, Optimization false, Expressions true, Deforming true
(8 rows)
```

在这种情况下，我们得到了5个工作者（就像预期的那样）。

然而，在有些情况下，你会希望某个表所使用的核心数量要多得多。试想一下，一个200GB的数据库，1TB的内存，而只有一个用户。这个用户可以用完所有的CPU，而不会对其他人造成伤害。ALTER TABLE可以用来推翻我们刚才讨论的内容。

```
test=# ALTER TABLE t_parallel SET (parallel_workers = 9);
ALTER TABLE
```

如果你想推翻x3规则来决定所需的CPU数量，你可以使用ALTER TABLE来明确地硬编码CPU的数量。 注意，max_parallel_workers_per_gather仍将有效，并作为上限。 如果你看一下计划，你会发现实际上会考虑核心的数量（看一下worker计划就知道了）。

```
test=# explain SELECT count(*) FROM t_parallel;
 QUERY PLAN
------------------------------------------------------------------------------
---------------
 Finalize Aggregate (cost=146343.32..146343.33 rows=1 width=8)
 -> Gather (cost=146342.39..146343.30 rows=9 width=8)
 Workers Planned: 9
 -> Partial Aggregate (cost=145342.39..145342.40 rows=1 width=8)
 -> Parallel Seq Scan on t_parallel (cost=0.00..138397.91 rows=2777791 width=0)
 JIT:
 Functions: 4
 Options: Inlining false, Optimization false, Expressions true, Deforming true
(8 rows)
Time: 2.454 ms
```

然而，这并不意味着这些核心也被实际使用。

```
test=# explain analyze SELECT count(*) FROM t_parallel;
 QUERY PLAN
------------------------------------------------------------------------------
-------
 Finalize Aggregate (cost=146343.32..146343.33 rows=1 width=8)
 (actual time=1375.606..1375.606 rows=1 loops=1)
 -> Gather (cost=146342.39..146343.30 rows=9 width=8)
 (actual time=1374.411..1376.442 rows=8 loops=1)
 Workers Planned: 9
 Workers Launched: 7
 -> Partial Aggregate (cost=145342.39..145342.40 rows=1 width=8)
 (actual time=1347.573..1347.573 rows=1 loops=8)
 -> Parallel Seq Scan on t_parallel (cost=0.00..138397.91 rows=2777791 width=0)
 (actual time=0.049..844.601 rows=3125000 loops=8)
 Planning Time: 0.028 ms
 JIT:
 Functions: 18
 Options: Inlining false, Optimization false, Expressions true, Deforming true
 Timing: Generation 1.703 ms, Inlining 0.000 ms, Optimization 1.119 ms,
 Emission 14.707 ms, Total 17.529 ms
 Execution Time: 1164.922 ms
(12 rows)
```

正如你所看到的，尽管计划有九个进程，但只有七个核心被启动。 这其中的原因是什么呢？在这个例子中，还有两个变量起了作用。

```
test=# SHOW max_worker_processes;
 max_worker_processes
----------------------
 8
(1 row)
test=# SHOW max_parallel_workers;
 max_parallel_workers
----------------------
 8
(1 row)
```

第一个进程告诉PostgreSQL一般有多少个工作进程。 max_parallel_workers说明有多少个工作进程可用于并行查询。为什么有两个参数？后台进程不仅仅是由并行查询基础设施使用的--它们还可以用于其他目的，因此大多数开发者决定使用两个参数。 一般来说，我们Cybertec（https://www.cybertec-postgresql.com）倾向于将max_worker_processes设置为服务器中CPU的数量。似乎使用更多通常没有好处。

7.1 PostgreSQL 能够并行做什么?

正如我们在本节中已经提到的，从PostgreSQL 9.6开始，对并行的支持已经逐渐得到改善。在每个版本中，都会增加新的东西。

下面是最重要的可以并行的操作。

- 并行顺序扫描
- 并行索引扫描(仅btrees)
- 并行位图堆扫描
- 并行连接(所有类型的连接)
- 并行btree创建(CREATE INDEX)
- 并行聚合
- 并行附加
- VACUUM
- CREATE INDEX

在PostgreSQL 11中，已经增加了对并行索引创建的支持。普通的排序操作还不能完全并行--到目前为止，只有btree的创建可以并行进行。为了控制并行的数量，我们需要应用以下参数。

```
test=# SHOW max_parallel_maintenance_workers;
 max_parallel_maintenance_workers
----------------------------------
 2
(1 row)
```

并行的规则基本上与正常操作的规则是一样的。

如果你想加快索引的创建速度，可以考虑看看我的一篇与索引创建和性能有关的博文：https://www.cybertec-postgresql.com/en/postgresql-parallel-create-index-forbetter-performance/。

7.2 实践中的并行性

现在我们已经介绍了并行性的基本知识，我们必须了解它在现实世界中的意义。让我们看一下下面的查询。

```
test=# explain SELECT * FROM t_parallel;
 QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on t_parallel (cost=0.00..360621.20 rows=25000120 width=4)
(1 row)
```

为什么PostgreSQL不使用并行查询？表足够大，而且PostgreSQL工作者可以使用，那么为什么不使用并行查询？答案是，进程间的通信真的很昂贵。如果PostgreSQL必须在进程之间运送行，那么查询实际上会比单进程模式下慢。优化器使用成本参数来惩罚进程间通信。

```
#parallel_tuple_cost = 0.1
```

每当一个元组在进程之间移动时，0.1分将被添加到计算中。为了看看PostgreSQL在被迫的情况下如何运行并行查询，我包括了下面这个例子。

```
test=# SET force_parallel_mode TO on;
SET
test=# explain SELECT * FROM t_parallel;
 QUERY PLAN
------------------------------------------------------------------------------
 Gather (cost=1000.00..2861633.20 rows=25000120 width=4)
 Workers Planned: 1
 Single Copy: true
 -> Seq Scan on t_parallel (cost=0.00..360621.20 rows=25000120 width=4)
(4 rows)
```

正如你所看到的，成本要比单核模式高。在现实世界中，这是一个重要的问题，因为很多人会想知道为什么PostgreSQL要采用单核。 在一个真实的例子中，同样重要的是要看到，更多的核心并不自动导致更多的速度。 要找到完美的内核数量，需要一个微妙的平衡行为。

8.引入即时编译（JIT）

JIT编译一直是PostgreSQL 11的热门话题之一。它是一项重要的工作，而且最初的结果看起来很有希望。然而，让我们从基本原理开始：JIT编译是怎么回事？当你运行一个查询时，PostgreSQL必须在运行时弄清很多东西。 当PostgreSQL本身被编译时，它不知道你接下来会运行哪种查询，所以它必须为各种情况做好准备。

核心是通用的，这意味着它可以做各种各样的事情。然而，当你在一个查询中，你只想尽可能快地执行当前的查询--而不是其他一些随机的东西。关键是，在运行时，你对你要做的事情的了解要比在编译时（也就是PostgreSQL被编译时）多得多。这正是重点：当启用JIT编译时，PostgreSQL将检查你的查询，如果它恰好足够耗时，将为你的查询创建高度优化的代码（及时性）。

8.1 配置 JIT

要使用JIT，必须在编译时添加。以下是可用的配置选项。

```
 --with-llvm build with LLVM based JIT support
...
 LLVM_CONFIG path to llvm-config command
```

一些Linux发行版提供了一个包含对JIT支持的额外软件包。如果你想使用JIT，请确保这些软件包已经安装。

一旦你确定JIT是可用的，下面的配置参数将是可用的，以便您可以为您的查询微调 JIT 编译：

```
#jit = on # allow JIT compilation
#jit_provider = 'llvmjit' # JIT implementation to use
#jit_above_cost = 100000 # perform JIT compilation if available
 # and query more expensive, -1 disables
#jit_optimize_above_cost = 500000 # optimize JITed functions if query is
 # more expensive, -1 disables
#jit_inline_above_cost = 500000 # attempt to inline operators and
 # functions if query is more expensive,
 # -1 disables
```

jit_above_cost意味着只有当预期成本至少为100,000单位时才考虑JIT。为什么会有这种情况呢？如果一个查询不够长，编译的开销会比潜在的收益高很多。因此，只能尝试进行优化。然而，还有两个参数：如果查询被认为比500,000个单位更昂贵，就会尝试真正的深度优化。

在这种情况下，函数调用将被内联。 在这一点上，PostgreSQL只支持低级虚拟机（LLVM）作为一个JIT后端。 也许将来也会有其他的后端。目前，LLVM做得非常好，涵盖了专业背景下使用的大多数环境。

8.2 运行查询

为了向你展示JIT是如何工作的，我们将编译一个简单的例子。让我们从创建一个大表开始--一个包含大量数据的表。记住，JIT编译只有在操作足够大时才有用。对于初学者来说，5000万行应该就足够了。下面的例子显示了如何填充表。

```
jit=# CREATE TABLE t_jit AS
 SELECT (random()*10000)::int AS x, (random()*100000)::int AS y,
(random()*1000000)::int AS z
 FROM generate_series(1, 50000000) AS id;
SELECT 50000000
jit=# VACUUM ANALYZE t_jit;
VACUUM
```

在这种情况下，我们将使用随机函数来生成一些数据。为了向你展示JIT是如何工作的，并使执行计划更容易阅读，你可以关闭并行查询。JIT在并行查询时工作正常，但执行计划往往要长很多。

```
jit=# SET max_parallel_workers_per_gather TO 0;
SET
jit=# SET jit TO off;
SET
jit=# explain (analyze, verbose) SELECT avg(z+y-pi()), avg(y-pi()), max(x/pi())
 FROM t_jit
 WHERE ((y+z))>((y-x)*0.000001);
 QUERY PLAN
------------------------------------------------------------------------------
----------------------
 Aggregate (cost=1936901.68..1936901.69 rows=1 width=24)
 (actual time=20617.425..20617.425 rows=1 loops=1)
 Output: avg((((z + y))::double precision - '3.14159265358979'::double
precision)),
 avg(((y)::double precision - '3.14159265358979'::double precision)),
 max(((x)::double precision / '3.14159265358979'::double precision))
 -> Seq Scan on public.t_jit (cost=0.00..1520244.00 rows=16666307 width=12)
 (actual time=0.061..15322.555 rows=50000000 loops=1)
 Output: x, y, z
```

```
 Filter: (((t_jit.y + t_jit.z))::numeric > (((t_jit.y - t_jit.x))::numeric *
0.000001))
 Planning Time: 0.078 ms
 Execution Time: 20617.473 ms
(7 rows)
```

在本例中，查询耗时 20 秒。

> 我使用了一个VACUUM函数，以确保所有的提示位等都被正确设置，以保证JIT查询和正常查询之间的公平比较。

让我们在启用 JIT 的情况下重复此测试：

```
jit=# SET jit TO on;
SET
jit=# explain (analyze, verbose) SELECT avg(z+y-pi()), avg(y-pi()), max(x/pi())
 FROM t_jit
 WHERE ((y+z))>((y-x)*0.000001);
 QUERY PLAN
------------------------------------------------------------------------------------
---------------------
 Aggregate (cost=1936901.68..1936901.69 rows=1 width=24)
 (actual time=15585.788..15585.789 rows=1 loops=1)
 Output: avg((((z + y))::double precision - '3.14159265358979'::double
precision)),
 avg(((y)::double precision - '3.14159265358979'::double precision)),
 max(((x)::double precision / '3.14159265358979'::double precision))
 -> Seq Scan on public.t_jit (cost=0.00..1520244.00 rows=16666307 width=12)
 (actual time=81.991..13396.227 rows=50000000 loops=1)
 Output: x, y, z
 Filter: (((t_jit.y + t_jit.z))::numeric > (((t_jit.y - t_jit.x))::numeric *
0.000001))
 Planning Time: 0.135 ms
 JIT:
 Functions: 5
 Options: Inlining true, Optimization true, Expressions true, Deforming true
 Timing: Generation 2.942 ms, Inlining 15.717 ms, Optimization 40.806 ms,
 Emission 25.233 ms,
 Total 84.698 ms
 Execution Time: 15588.851 ms
(11 rows)
```

在这种情况下，你可以看到查询的速度比以前快了很多，这已经很重要了。在某些情况下，好处甚至可以更大。然而，请记住，重新编译代码也与一些额外的工作有关，所以它对每一种查询都没有意义。

了解PostgreSQL的优化器对提供良好的性能是非常有益的。深入研究这些主题以确保良好的性能是有意义的。

9 总结

在这一章中，我们讨论了一些查询优化。你了解了优化器和各种内部优化，如常量折叠、视图内联、连接等等。 所有这些优化都有助于实现良好的性能，并有助于大大加快事情的进展。

现在我们已经介绍了优化的情况，在下一章，即第7章，编写存储过程，我们将谈论存储过程。你将了解到PostgreSQL的所有选项，我们可以用这些选项来处理用户定义的代码。